NANYANG TECHNOLOGICAL UNIVERSITY

Structural edge detection of photographic images of rooms with machine learning

Yao Xin Meng

School of Mechanical and Aerospace Engineering

Nanyang Technological University

2019/2020

Abstract

In the task to reconstruct 3D models of room architecture from photographic images, identifying the relevant structural edges of the room amidst the noise has been a tremendous challenge. This Final Year Project sets out to determine if machine learning can be a viable alternative to classical edge-detection algorithms and, if so, determine the machine learning model that has the best performance.

The methodology for this project involves four main parts – generating a labelled dataset, augmenting the labelled data to enlarge the dataset, processing the dataset, and training various models on the dataset. For this project, training is carried out on four different Fully Convolutional Network (FCN) architectures, namely SegNet, U-Net, DenseNet and a pre-trained FCN-RESNET101. For each model, the input is an RGB image and the output is a greyscale image with each pixel indicating its probability of not laying on a relevant edge.

The results obtained from the training indicated that the model based on U-Net had the best performance out of the four. Using this finding, further finetuning of the U-Net model's parameters and hyperparameters are performed to further enhance its performance. Post-processing such as edge-thinning and feature extraction is applied to the output of the final model to obtain the line equation of every predicted edge.

The results obtained showed strong promise in discerning structural edges, thus validating the initial hypothesis that machine learning is a viable alternative to classical algorithms. Future works include further enhancing the current model's accuracy and creating an algorithm to construct a wireframe model from the line equations.

Acknowledgements

I would like to express my gratitude towards these individuals, without whom this project could not have been successful.

My Final Year Project mentor, Associate Professor Lee Yong Tsui, for his continuous guidance and support throughout the year.

The staff at the MAE Computer Aided Engineering Laboratory 1, for their patience and assistance.

My friends, Howard, Ada and Yufei, for never failing to entertain my dumb questions.

And, my family, for their unyielding love and support.

Table of Contents

1. Introduction

1.1 Background and Motivation

This project aims to reconstruct the 3D model of room architecture from photographic images. This has many applications, such as measuring room dimensions for interior design or inspection of real estate. The process of 3D reconstruction from photographs can be broken down into 4 steps:

1. Extracting the relevant structural edges
2. Determining the line equation of each edge
3. Using the line equations to form a wireframe model
4. Creating the 3D model from the wireframe model

The greatest challenge in this process is the first step – extracting the relevant edges in photographs. Due to the wide variety of noise and objects present in photographs, a deterministic approach to the problem, such as the Canny edge finder (Canny, 1986), proved to be inadequate in past projects (Figure 1). This is because classic

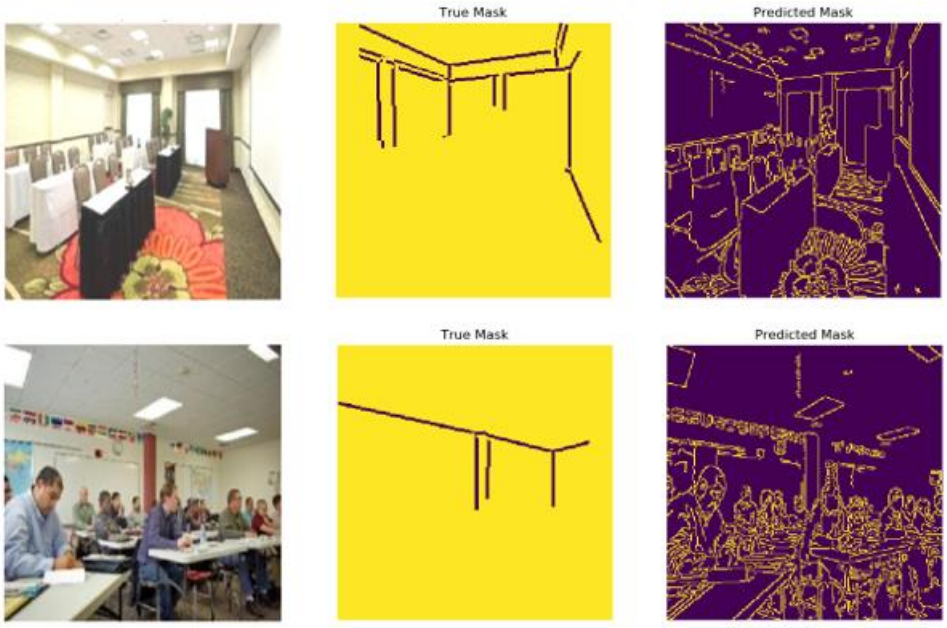


Figure 1 – Results from Canny Edge Detector (OpenCV)

algorithms are indiscriminate and fail to distinguish between different types of edges, therefore it is unable to filter out the irrelevant edges.

1.2 Objectives and Scope

Due to the limitations of classic algorithms, machine learning is proposed as a possible alternative for detecting structural edges in photographic images. It is hypothesised that a neural network can be trained to extract only the relevant edges from photographic images.

Therefore, the objective of this project is to establish whether machine learning is a viable solution to the challenge of extracting structural edges in photographs and, if so, determine the neural network that is most suitable for performing this task.

1.3 Layout of Report

In Chapter 1, the background and motivation for this project are presented and the objectives and scope are reviewed.

In Chapter 2, literature used for this project is reviewed and the concept of Fully Convolutional Networks is briefly introduced. In addition, some of the existing technologies that were referenced and used for this project are briefly discussed.

In Chapter 3, the methodology used in every stage of this project, from the data labelling, data augmentation and data processing to the training of each neural network model, will be discussed. The motivation for choosing the methodology in each step is also discussed.

In Chapter 4, the results obtained from the neural network training is presented and analysed. The final model architecture is justified based on these results and the final desired output from this model is presented.

In Chapter 5, the report will end off with a summary of the entire project and propose some of the future work that can be done to further this project.

2. Literature Review

2.1 Overview

Recent developments in machine learning have mainly revolved around deep learning and it represents an opportunity for machines to perform tasks that have traditionally been impossible (Marr, 2018). Of particular interest to this project are Convolutional Neural Networks (CNN) and Fully Convolutional Networks (FCN), both of which have led to huge improvements in image processing capabilities. FCNs are a natural progression from CNNs and they differ from CNNs in their last few layers. While CNNs generally have a linear output layer for classifying an entire image, FCNs have a convolutional output layer that attempts to classify each pixel in an image (Jonathan Long, 2014). For example, an image of cats, duck and dog as shown below will generate two different outputs using a CNN and a FCN.

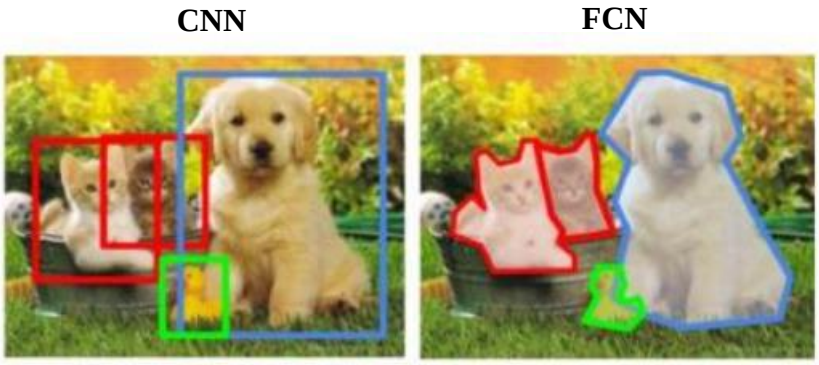


Figure 2 – Capabilities of CNN vs FCN

This has caused FCNs to rapidly gain popularity as it can potentially identify multiple objects in a single image as well as determine the location, size and shape of each object. This process is known as image segmentation and has applications in many fields, such as recognition of road features (traffic lights, pedestrian crossing, road markings, etc.) in autonomous vehicles.

2.2 Fully Convolutional Networks

Most FCNs contain the same few basic layers as building blocks. The most common layers used in 2D image processing are described in Table 1.

Layer	**Description**
2D Convolution	Creates a specified number of kernels that each performs convolution on the image. Convolution is a form of matrix operation that calculates the sum of the element-wise multiplication between the input matrix and the kernel matrix.
Transposed 2D Convolution	This performs the inverse of a 2D convolution and is often used for up-sampling.
Max Pool 2D	Creates a filter of specified N x N dimensions. The filter traverses the input and takes the largest value in each N x N region of the input.
Max Unpool 2D	Performs the inverse a Max Pool 2D. However, the non-maximal values are lost in the Max Pool process, hence they are replaced with zeros instead.
Batch Normalisation	Normalises each batch of training data to improve the training process.
ReLU	It is called a Rectified Linear Unit and is an activation function. Mathematically, it is represented by $f(x) = \max(0, x)$.
Leaky ReLU	This is a variation of ReLU and introduces a small positive gradient for negative values. Mathematically, it is represented by $f(x) = \max(0, x) + 0.01 \times \min(0, x)$.

Table 1 – Descriptions of common layers used in FCNs

There are a few notable FCN architectures that are readily available. The more successful ones include SegNet, U-Net and DenseNet (Shah, 2017). More details regarding each of these architectures is provided below.

1. SegNet architecture (Vijay Badrinarayanan, 2015)

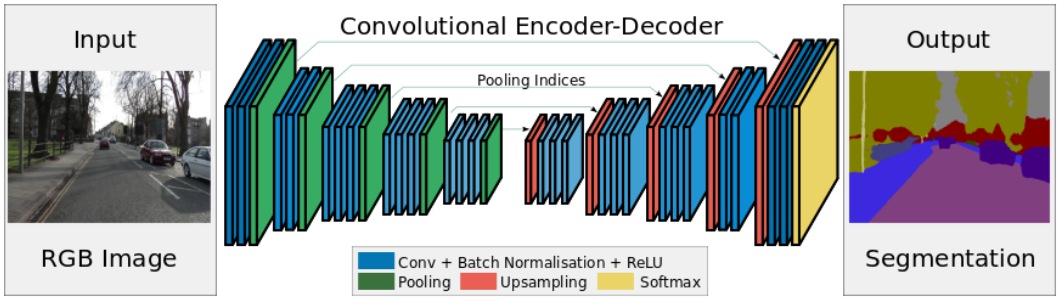


Figure 3 – SegNet architecture

SegNet uses cascading modules of encoders followed by cascading modules of decoders to extract key features from an image. Encoding modules consist of Convolution, Batch Normalization, ReLU (activation function) and Max Pool, while decoding modules consist of Max Unpool, Convolution, Batch Normalization and ReLU.

2. U-Net architecture (Olaf Ronneberger, 2015)

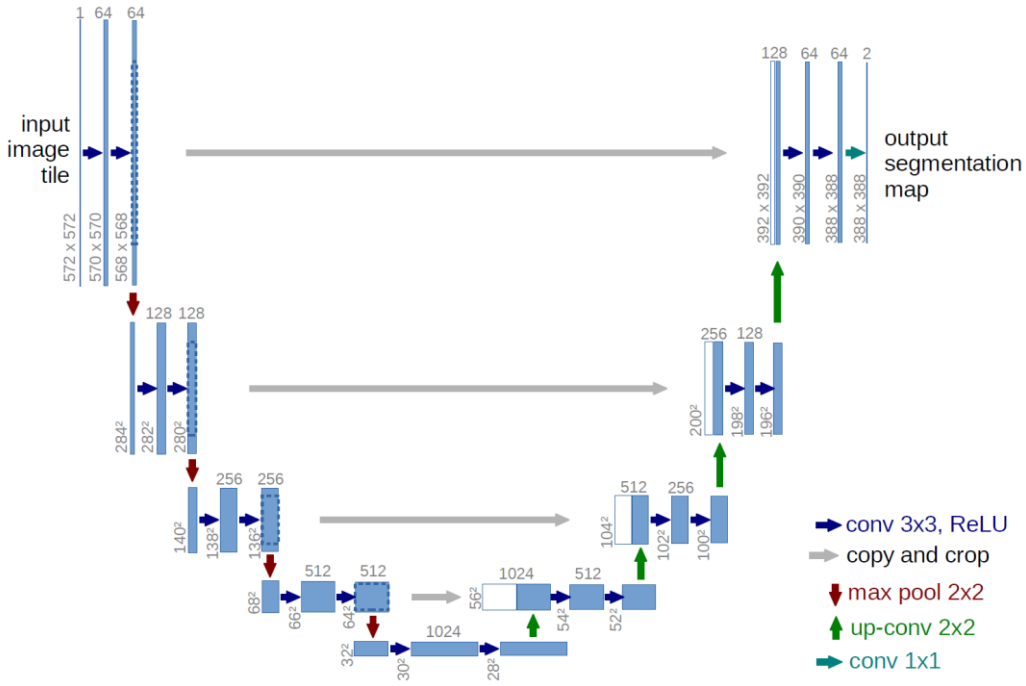


Figure 4 – U-Net architecture

U-Net is similar to SegNet in many ways, but there are 2 key differences. First, it uses Transposed Convolution instead of Max Unpool for up-sampling. This creates additional parameters for the model to train on. Second, the output from each encoder block is cloned and concatenated onto the input of the corresponding decoder block. This allows some features that might otherwise have been lost in the encoding process to be captured by the decoder blocks.

3. DenseNet architecture (Simon Jégou, 2016)

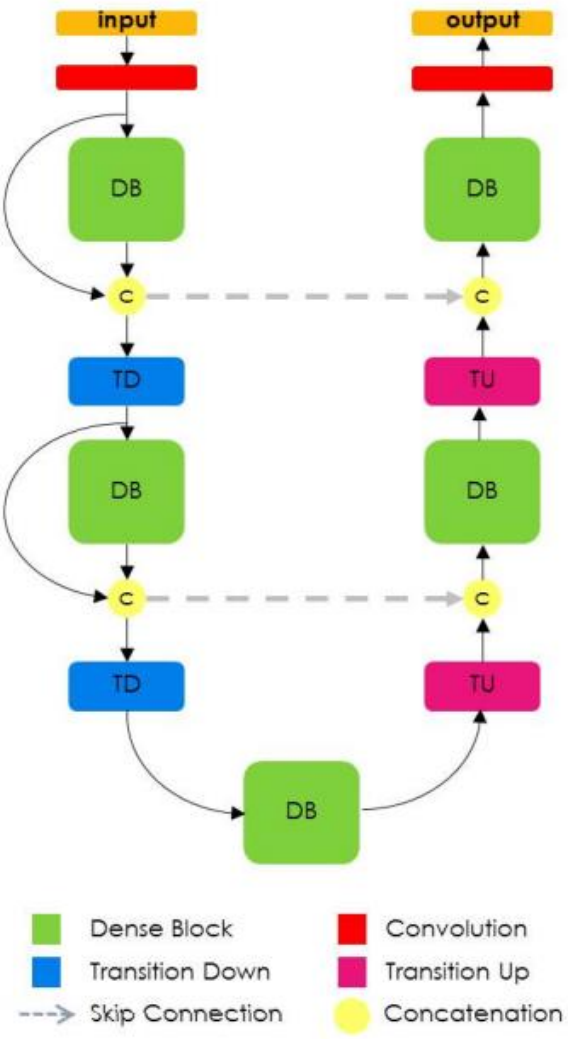


Figure 5 – DenseNet architecture

DenseNet is similar to U-net in that it also uses clone and concatenation to allow some features to skip some layers. However, the main difference is that its building block is the Dense Block (Figure 6). The Dense Block (DB) is made by concatenating each input with the output of the next layer and concatenating the outputs of all 4 layers together at the end.

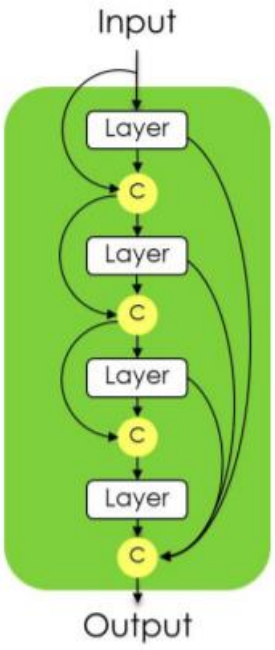


Figure 6 – A Dense Block in the DenseNet architecture

4. Transfer Learning using pre-trained FCN-RESNET101

FCN-RESNET101 is a Fully Convolutional Network model with a ResNet-101 backbone. The model has been pretrained on a subset of the Common Objects in Context (COCO) 2017 dataset. The pretrained weights were used and only the last few layers of the model were modified and retrained for the purposes of this project.

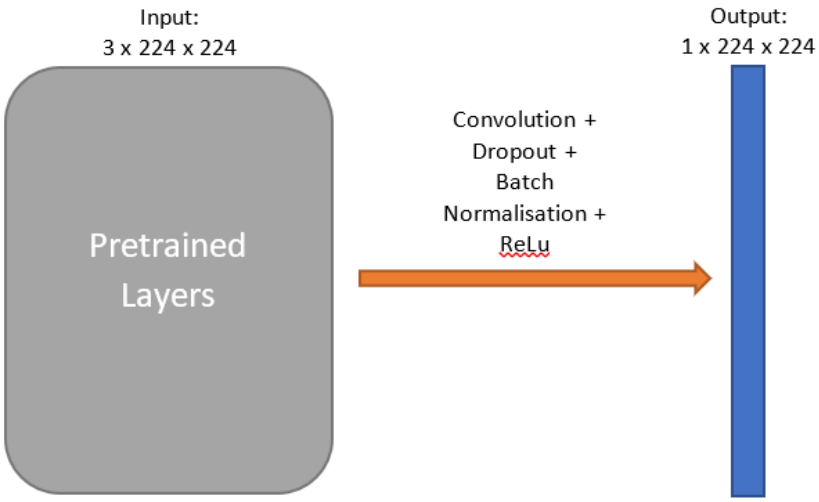


Figure 7 – Transfer Learning model using FCN-RESNET101 architecture

Testing is required to determine which architecture would work best for the purposes of this project since it was difficult to determine at first glance. For practical purposes, only SegNet, U-Net, DenseNet and FCN-RESNET101 were chosen as candidates. In order to implement these architectures, a suitable language and library must first be chosen.

2.3 Machine Learning Libraries

Most machine learning programs are written in Python due to its simplicity and the tremendous amount of open source libraries and resources supporting the language. Therefore, Python is the programming language of choice for this project. In recent years, 2 machine learning libraries that are built upon Python have become widely popular:

1. TensorFlow – developed by Google
2. PyTorch – developed by Facebook

While each library has its strengths and weaknesses, TensorFlow was initially chosen for this project as it was the preferred library in the industry and was also well supported with comprehensive documentation and forums (Jain, 2018). However, PyTorch was eventually favoured as it was a lot more memory-efficient in implementing the neural network, a condition that was necessary due to hardware constraints. In addition, it was more intuitive than TensorFlow when implementing non-sequential neural networks, such as U-Net and DenseNet. This will be further elaborated on in the Methodology section.

2.4 Other Libraries

This project deals with photographic images and it is inevitable that image processing capabilities are required at various stages of this project. For these purposes, this project tapped on the Open Source Computer Vision Library (OpenCV) for its vast selection of image processing algorithms. OpenCV is BSD-licensed, therefore it is free for businesses and individuals to use and modify its code (OpenCV, 2020).

In addition, this project requires extensive image visualisation as well. The visualisation tool should be capable of interpreting a wide variety of data such as the standard RGB 8-bit unsigned integer format, or a greyscale 32-bit floating point format that is more commonly used in machine learning. As such, Matplotlib, an open source library in Python, was used for this project (Hunter, 2007).

Another tool that was used extensively for this project was Pygame, a library built using Python for users to write video games and is compatible on most systems (Pygame, 2020). It was used to create a custom labelling tool and develop an intuitive Graphic User Interface to simplify the process of data labelling.

3. Methodology

3.1 Overview

The machine learning model chosen for this project is the Fully Convolutional Networks (FCN). FCNs are widely used for image processing due to its ability to segment an image into categories. The input to the model is a 256 x 256 RGB image and the expected output is a 256 x 256 grayscale “mask” containing the predicted structural edges of the original image.

The process of training the model is divided into four parts:

1. Labelling of data
2. Data Augmentation
3. Data Processing
4. Training the Model

The images that were used were sourced from a scene-centric database used in the LSUN Scene Classification Challenge (Yu, et al., 2015). Approximately 5,000 images from categories “bedroom”, “classroom”, “conference room” and “living room” were selected as the dataset for this project (Figure 8). The rationale for choosing images across multiple categories was to increase the variety of objects in and layout of the room depicted in each image. This reduces the likelihood of overfitting and improves the model’s ability to recognise rooms that are not covered in the dataset, such as storerooms or offices.

Figure 8 – Clockwise from top left: classroom, bedroom, conference room, living room

3.2 Data Labelling

Data labelling was done by tracing structural edges in each image. However, popular online tools such as MIT LabelMe (Russell, Torralba, Murphy, & Freeman, 2008) and Amazon Mechanical Turk (Amazon, 2018) did not have the capabilities suitable for this task. For example, LabelMe only contains polygon tools and is suitable for creating masks over objects (Figure 9).

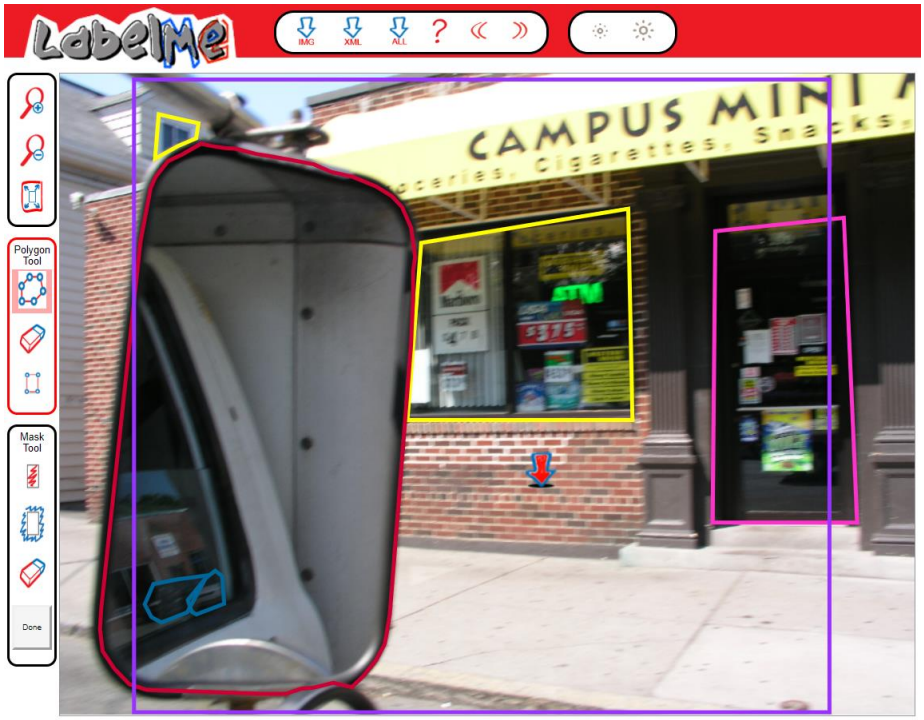


Figure 9 – Image labelling using MIT LabelMe tool

This project, on the other hand, requires masks over edges, hence a line tool would be more suitable. With these considerations, a custom labelling tool was built using Python and the Pygame library (Figure 10).

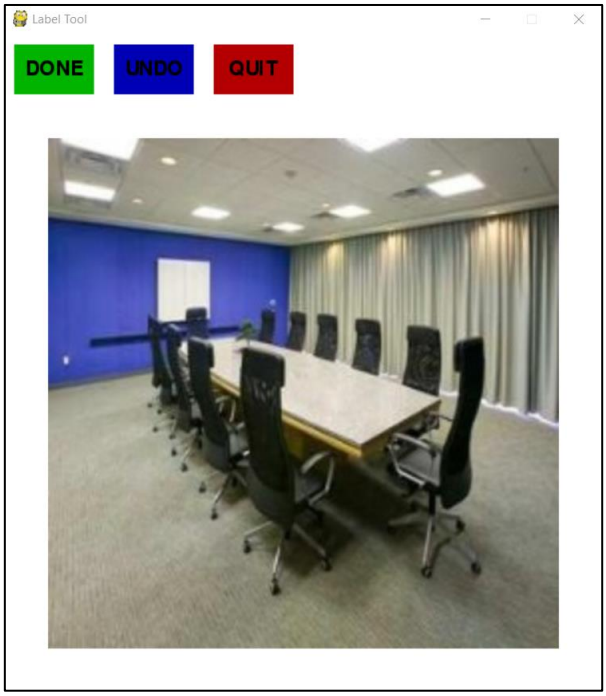


Figure 10 – Custom Label Tool

Essential to every machine learning model is its loss function, which is a measure of the error between the model output and the target value. For this project, the loss function should measure the error of each pixel. Given that edges make up very few pixels in any given image, one concern is that the optimised model might get trapped in a local minimum and generate blank images. In order to prevent this, it is proposed that a larger error can be artificially introduced. One way of doing this is to create a heatmap around each edge (Figure 11). Evidently, the error of a blank model output (an array of '1's) will be much larger with the heatmap, hence reducing the likelihood of such an output. Another proposed idea was to simply thicken the edge from a width of 1 pixel to 3 pixels.

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	1
1	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	1
1	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	1
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	1
1	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	1
1	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	1
1	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 11 – Example of Heatmap Feature Around an Edge

Since the labelling tool was custom-built, these modifications to the edges could be easily built into the tool, hence automating this process. An example of how the heatmap feature was implemented as a function is as follows:

```
def heatmap(drawing, point0, point1):
      # get line vector between the 2 points
      vector = subtract(point0, point1)

      # get vertical and horizontal components of vector
      ver = dot(vector, [0,1])
      hor = dot(vector, [1,0])

      # more horizontal than vertical
      # create grey lines above and below original line
      if abs(ver) < abs(hor):
            draw(drawing, point0, point1, [0,1], grey)
            draw(drawing, point0, point1, [0,-1], grey)
            # and other shades of grey lines

      # more vertical than horizontal
      # create grey lines left and right of original line
      else:
            draw(drawing, point0, point1, [1,0], grey)
            draw(drawing, point0, point1, [-1,0], grey)
            # and other shades of grey lines

def draw(drawing, point0, point1, offset, fill):
      new0 = add(point0, offset)
      new1 = add(point1, offset)
      # this is a function from the ImageDraw library
      # draws a line from p0 to p1 with fill=fill and width=width
      line([new0, new1], fill=fill, width=1)
```

Based on the considerations discussed above, a total of 3 different types of edges were used – 1 pixel edge, 3 pixel edge and heatmap edge (Figure 12).

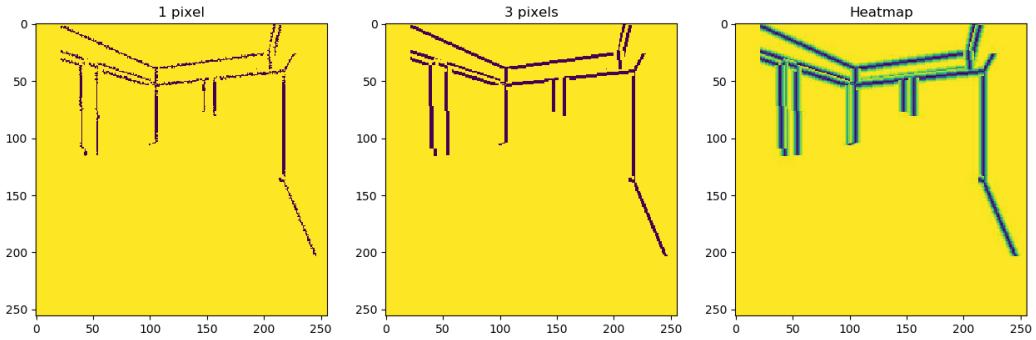


Figure 12 – Various edge types

3.3 Data Augmentation

While there is no ‘one size fits all’ answer for the recommended dataset size for machine learning, there are some general guidelines to follow. For regular machine

learning models such as regression, one can use the Vapnik-Chevronenkis dimension (Koiran & Sontag, 1998) to determine the minimum dataset size given the type of model used. However, this formula fails for deep learning models such as the kind used in this project. Specifically, traditional machine learning algorithms fails to achieve any significant improvements in performance beyond a certain dataset size, whereas performance increases logarithmically with greater training data size for deep learning models (Sun, Shrivastava, Singh, & Gupta, 2017). In other words, more data is always better for deep learning models. Based on this finding, it would be beneficial to increase the labelled dataset of 5,000 images as much as possible. To do so, common data augmentation techniques for images were used. Using the OpenCV library, all images are subjected to three augmentations (Figure 13):

1. A horizontal flip
2. An increase in brightness
3. A horizontal flip plus an increase in brightness

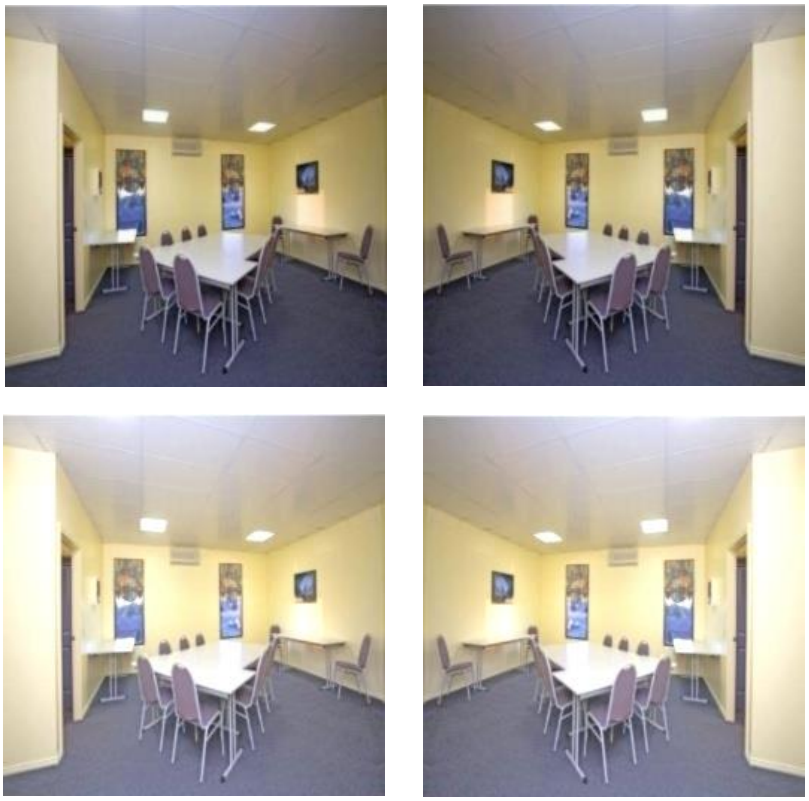


Figure 13 – Clockwise from top left: Original, Flipped, Brightened, Flipped + Brightened

The corresponding labels to the augmented image must also be altered (Figure 14):

1. A horizontal flip to match the flipped image
2. No change since brightness does not affect the location of structural edges
3. A horizontal flip to match the flipped image

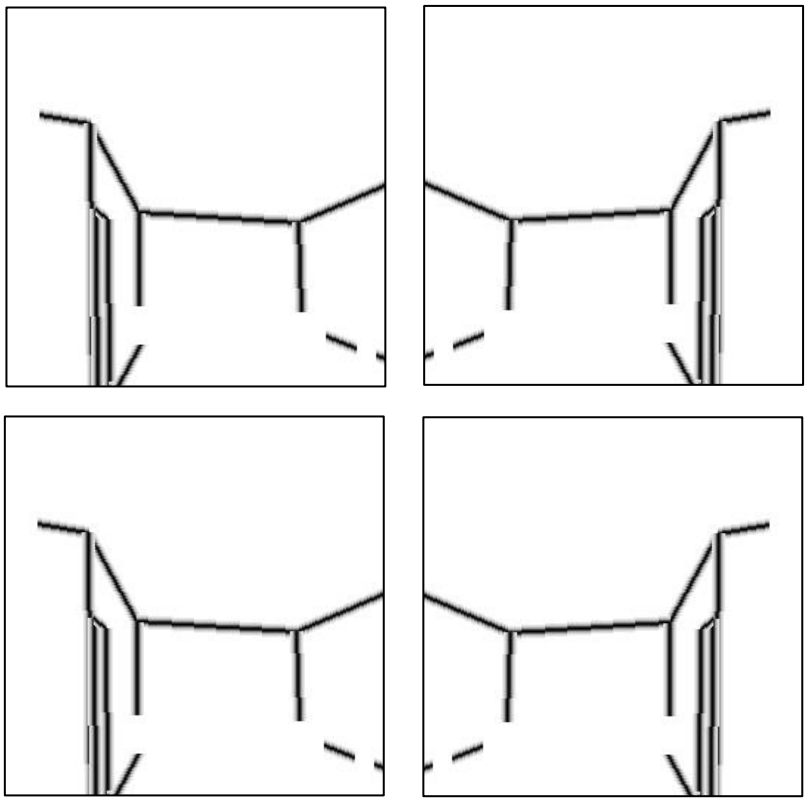


Figure 14 – Clockwise from top left: Original, Flipped, Brightened, Flipped + Brightened

Using these augmentation techniques, a total of 20,000 labelled images is contained within the dataset. The new dataset is then further split in a 7:3 ratio to obtain a training set and a test set. The 14,000 labelled images in the training set is used to train the various neural networks, while the 6,000 labelled images in the test set is used to validate the neural network and prevent overfitting.

3.4 Data Processing

Each pixel in an RGB image is an unsigned 8-bit integer and has a value range of [0, 255]. The first step in normalisation is to convert each pixel to a 32-bit float with a value range of [0, 1]. Subsequently, each RGB input image to the neural network is further normalized such that the entire dataset has a standard normal distribution. In theory, this helps to ensure that the gradient is uniform as it propagates across the

neural network, hence improving the convergence time (B, 2017). The detailed normalisation for each channel is as such:

$$output[channel] = \frac{input[channel] - mean[channel]}{standard\ deviation[channel]}$$

Each channel's mean and standard deviation is calculated based on the training dataset. Each image in the training dataset is assigned a value based on the mean value of its pixels. These values are then used to calculate mean and standard deviation of the entire dataset as shown below:

$$y_i[channel] = \frac{\sum_j^d x_j[channel]}{d}, \qquad d = \ number\ of\ pixels\ in\ an\ image$$

$$mean[channel] = \frac{\sum_i^N y_i[channel]}{N}, \qquad N = \ number\ of\ images\ in\ the\ dataset$$

$$standard\ deviation[channel] = \sqrt{\frac{\sum_i^N (y_i[channel] - mean[channel])^2}{N}}$$

The final values obtained for each channel in this dataset is given below:

$$mean[R] = 0.658, \qquad standard\ deviation[R] = 0.136$$

$$mean[G] = 0.600, \qquad standard\ deviation[G] = 0.138$$

$$mean[B] = 0.541, \qquad standard\ deviation[B] = 0.150$$

Even though a standard normal distribution should in theory improve the training process, it was not the case for this project. During the training, the losses consistently spiked at epoch 22 (Figure 15) and subsequent epochs were incomputable, perhaps due to an overly large gradient. It was speculated that the PyTorch graph might have been inherently unstable for the range of values that were used, and future work can

be done in determining the exact reason for this behaviour. Regardless, the second step of the normalisation of the dataset was eventually abandoned, hence each pixel simply had a value range of [0, 1].

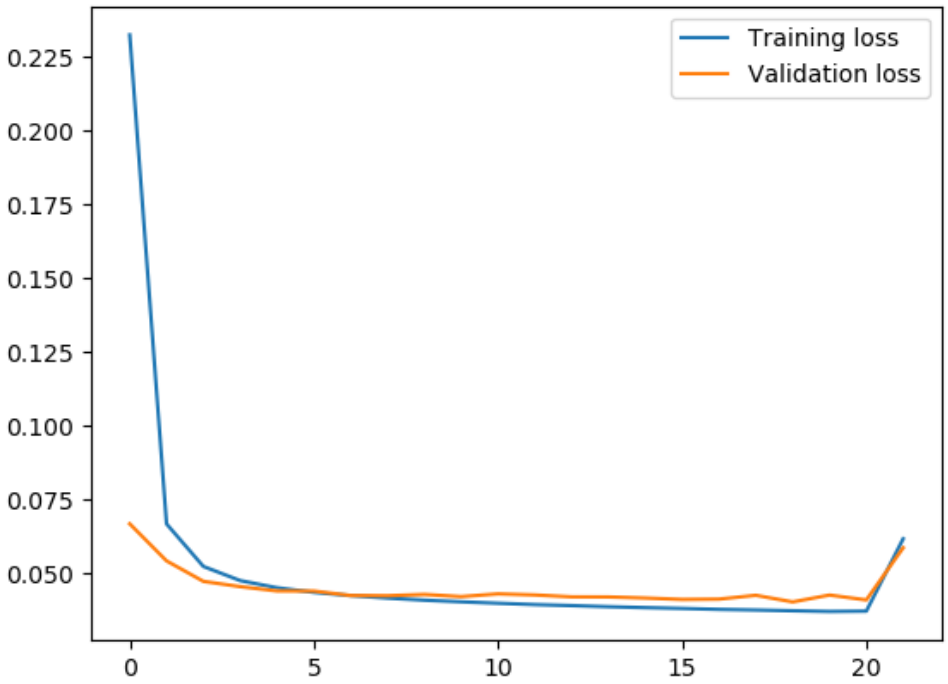


Figure 15 – Training Loss against epochs with standard normalised dataset

3.5 Training the Model

For each neural network, a suitable loss function must be chosen to measure the accuracy of the model. It was initially assumed that Mean Squared Error Loss (MSE) could be used as a loss function. MSE measures the deviance of the predicted value of each pixel from the true value. However, the training failed to yield any meaningful results. Upon further investigation, it was determined that MSE was only suitable for linear regression models, whereas this project uses image segmentation and is therefore considered a classification model.

As such, a loss function that measures the accuracy of the classification of each pixel would be suitable. To that end, 2 loss functions may be relevant – Cross Entropy Loss

and Binary Cross Entropy Loss. Cross Entropy Loss measures the performance of a classification model whose output is a probability value between 0 and 1 (Machine Learning Glossary, 2017). It is frequently used in multi-class classification problems. In PyTorch, this is implemented using the function CrossEntropyLoss and has the following equation (PyTorch, 2019):

$$\text{loss}(x, class) = -\log\left(\sum_j \frac{\exp(x[j])}{\exp\left(x[class]\right)}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

Binary Cross Entropy Loss is a special case of Cross Entropy Loss where the classification problem is strictly binary. In the case of edge detection, this is relevant since each pixel is either classified as an edge or not an edge. In PyTorch, this is implemented using the function BCEWithLogitsLoss and has the following equation (PyTorch, 2019):

$$\text{loss}(x, y) = \text{mean(L)}, \qquad L = \{l_1, \dots, l_n\}$$

$$l_n = -w_n[y_n \cdot \log\sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))]$$

$$\sigma(x) = Sigmoid(x) = \frac{1}{1 + \exp\left(-x\right)}$$

While both loss functions could theoretically work well for this classification problem, it was discovered that BCEWithLogitsLoss worked much better. The reason for this difference in performance is unknown and can be explored in future works. It is also important to note that BCEWithLogitsLoss has a Sigmoid activation function embedded within itself. As such, when running the model without a loss function (such as when obtaining the output mask from an input image), an additional Sigmoid function must be applied to the model output.

The optimisation function that was chosen for this project was Adam (Kingma & Ba, 2014). It is an algorithm for first-order optimization of objective functions and is suitable for large data and parameters. While there have been some concerns regarding its convergence (Bushaev, 2018), Adam remains one of the most popular optimisers for deep neural networks. An alternative optimisation function is the Stochastic Gradient Descent (SGD) and is similarly used for first-order optimization. For this project, SGD was not used due to time constraints, but it can be considered for future development. The models are each trained using Adam at a learning rate of 0.00001.

With these considerations, the four models based on SegNet, U-Net, DenseNet and transfer learning are trained on the training set for 50 epochs each. Each epoch attempts to fit each data in the training set exactly once. The model with the highest accuracy is then selected for further finetuning. The detailed results are highlighted in the next section.

3.6 Model Implementation with TensorFlow VS PyTorch

The deep learning models used in this project were initially implemented in TensorFlow rather than PyTorch because of its popularity in the industry, and its well-supported documentations and forums. The four models discussed above were implemented in TensorFlow and training was attempted. However, the model failed to be trained due to a lack of memory space even when it was run on a computer with 64GB RAM. As a result, the same model was implemented with PyTorch and training was successfully attempted.

The reason for this difference in memory requirements between the two machine learning frameworks lies in the way they define the computation graphs of models

(Jain, 2018). TensorFlow uses a static graph, which means that the entire computation graph is created before running the model. On the other hand, PyTorch uses a dynamic graph, hence it updates the graph on-the-go. Since the models that were used are deep neural networks with an extremely large number of parameters, the entire computation graph might exceed 64GB. Therefore, the model could not be trained with TensorFlow but worked fine with PyTorch. It is important to note that if more computational power was available, TensorFlow might have been superior to PyTorch because a static computation graph does not need to be constantly rebuilt.

4. Results

4.1 Training results

The four models based on SegNet, U-Net and DenseNet were each trained on 3 pixel wide edges over 50 epochs. The accuracy of the model is cross-validated with the test dataset and is summarised in Table 2. The loss is calculated using Binary Cross Entropy, while the evaluation metric is accuracy, which is calculated using absolute error as shown in the following equation:

$$Accuracy = 1 - \frac{\sum_i^N |y_{i,pred} - y_{i,true}|}{N}$$

Model	**Loss**	**Accuracy**	**Parameters Size (MB)[1]**
SegNet	0.0562	0.9616	8.75
U-Net	0.0506	0.9683	29.38
DenseNet	0.0436	0.9740	20.14
Transfer Learning	0.0252	0.9998	199.78

Table 2 – Summary of performance of various models

Evidently, SegNet is the most lightweight model, but it is also the least accurate. On the other hand, the transfer learning model based on FCN-RESNET101 scores the best in terms of both loss and accuracy, while U-Net is the most memory-intensive. Based on these results alone, it would seem the transfer learning model has the best performance. However, closer inspection at the visual output from the various models indicates that models that work well on paper may not be the best. As shown in Figure 16, the output from the transfer learning Model does not contain any recognisable

[1] Refers to the memory size of the entire neural network with fully trained weights

edges despite its stellar performance. In fact, U-Net has the best performance in identifying relevant structural edges.

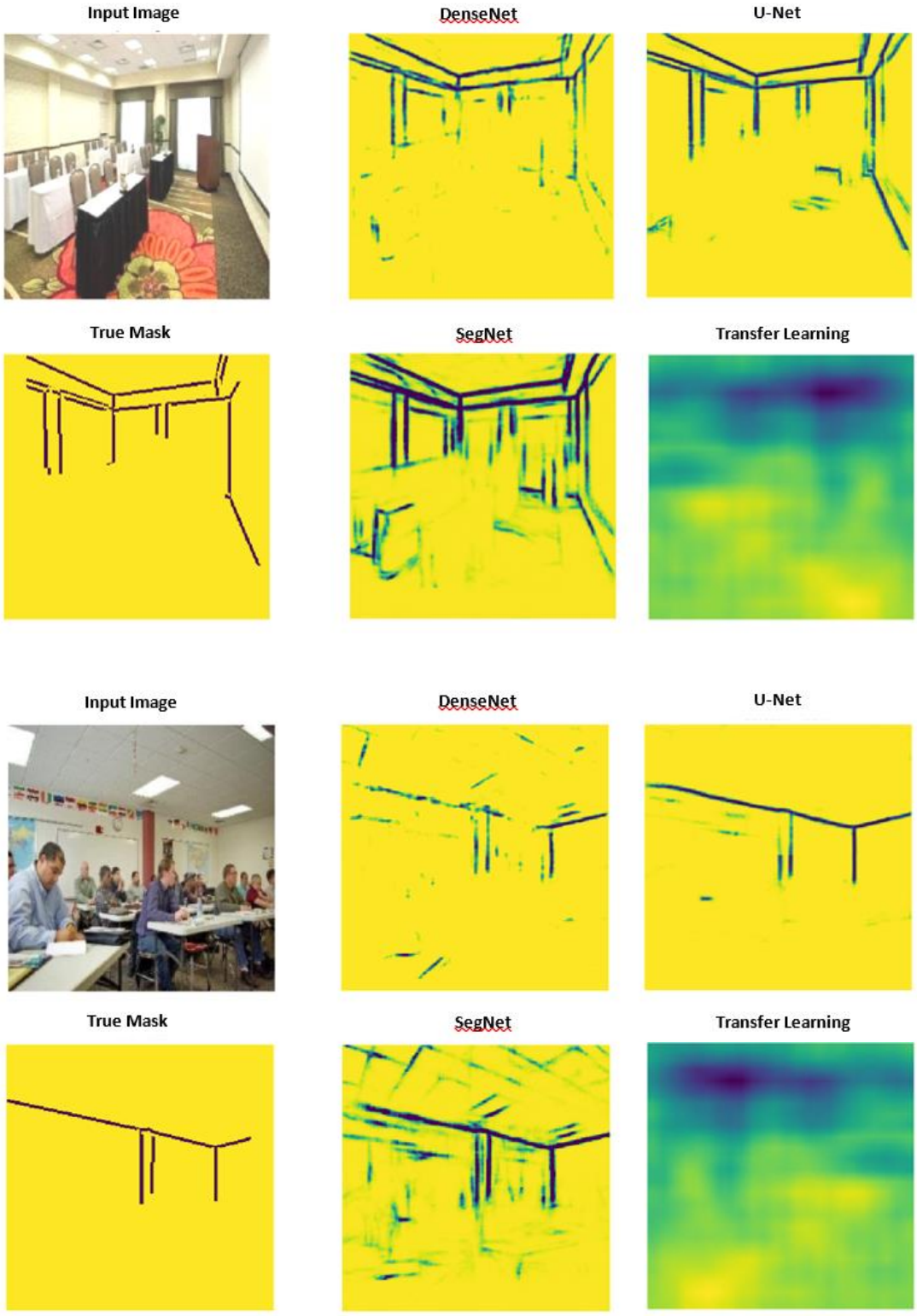


Figure 16 – Sample predicted masks from various models

There is one possible explanation for this phenomenon. The relation between the evaluation metric and the performance might be one of correlation rather than causation. In other words, a model with good performance will have a high accuracy, but a model with high accuracy may not necessarily perform well. Furthermore, the relation between the loss function and the evaluation metric is also one of correlation. As such, the loss function may also not be a reliable benchmark for performance. However, it is important to note that this explanation is purely hypothetical and leaves room for further studies to be done, especially in terms of finding a loss function and evaluation metric that are more reliable indicators of performance.

To further verify that the U-Net model is in fact extracting features from an image rather than force fitting onto the training set, a study of the output from each layer in the downsampling section of the model is carried out. Appendix A shows a visualisation of the convolutional outputs obtained from a sample image from the test set. A closer scrutiny reveals that the model is in fact extracting several pieces of information from the sample image, such as the walls, furniture and ceiling. This indicates that the U-Net model is extracting features properly, hence increasing the likelihood of it performing well even for images that do not resemble those in the training set.

As such, the final model that was used is based on U-Net and is summarised in Figure 17.

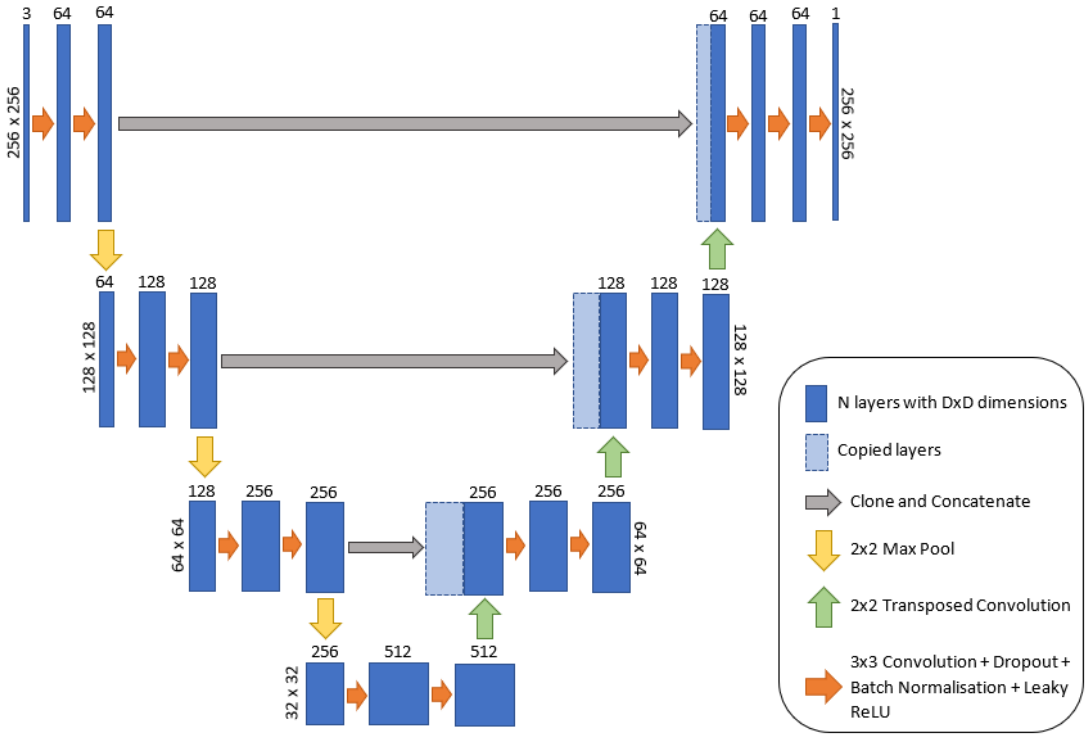


Figure 17 – Summary of model architecture

```
class Conv(Module):
      # define the architecture of this module
      def __init__(self, C_in, C_out):
            # C_in refers to number of input channels
            # C_out refers to number of output channels
            # n refers to percentage of neurons to drop out
            self.conv = Sequential(
                  Conv2d(C_in, C_out, kernel_size=3, stride=1,
                        padding=1),
                  Dropout2d(n),
                  BatchNorm2d(C_out),
                  LeakyReLU(),
                  Conv2d(C_out, C_out, kernel_size=3, stride=1,
                        padding=1),
                  Dropout2d(n),
                  BatchNorm2d(C_out),
                  LeakyReLU()
            )

      # forward pass of the module
      def forward(self, x):
            return self.conv(x)
```

The pseudocode snippet above shows an object that packages a layer in the U-Net model by using the PyTorch neural network library. This greatly improves the ease of

implementation. For example, the first layer in the final model could be created simply by invoking `Conv(3, 64)`.

4.2 Results from varying edge types

Further investigation was made into the effect of the type of edges used in the quality of the trained model. After training, the output of the model is processed using a threshold value of 0.5 to classify each pixel as either 0 or 1. The pseudocode snippet below shows how this process works as well as how the threshold can be easily tuned.

```
# global parameter
threshold = 0.5

# function to classify each pixel in mask
# 0 refers to an edge, 1 refers to a non-edge
def classify(mask):
    for x in mask:
        if x > threshold:
            x = 0
        else:
            x = 1
    return mask
```

Some sample results from testing the 1-pixel, 3-pixel and heatmap edges on the U-Net model is shown in Figure 18. From these results, it is determined that the model trained on 1-pixel edges has the best performance.

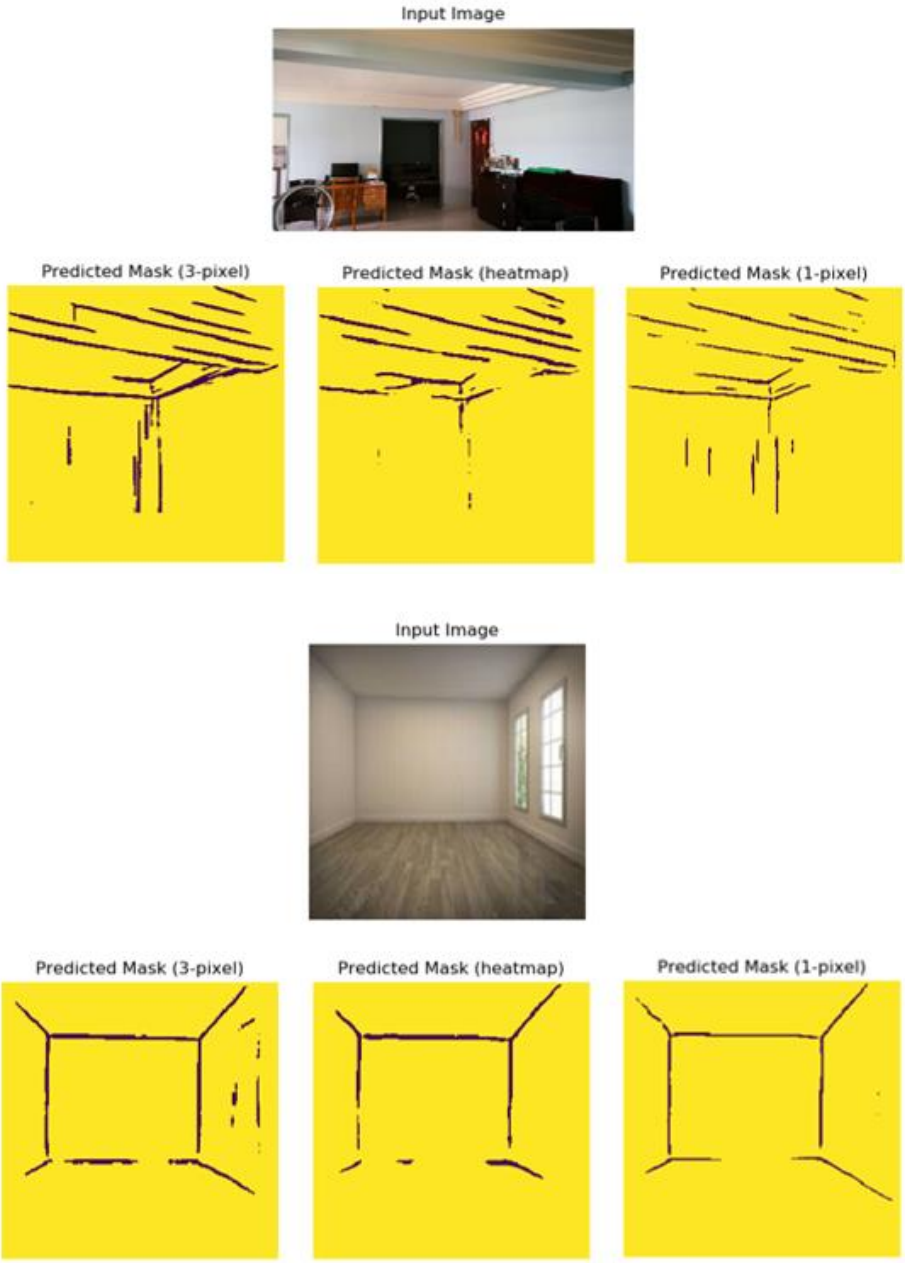


Figure 18 – Effect of edge type on model performance

4.3 Varying performance with epochs

Using this information, the U-Net model is then retrained on the 1-pixel edge dataset over 50 epochs to attain the ideal number of epochs for the model. The training and testing losses are plotted against epochs to identify the epoch number where testing losses are minimal (Figure 19). This is to prevent any overfitting of the model on the

trainset. From the results, the optimal number of epochs occurs around 34. Since the training set is shuffled, the loss obtained for each round of training would be slightly different. As such, a ballpark figure of epochs can be used to no detriment.

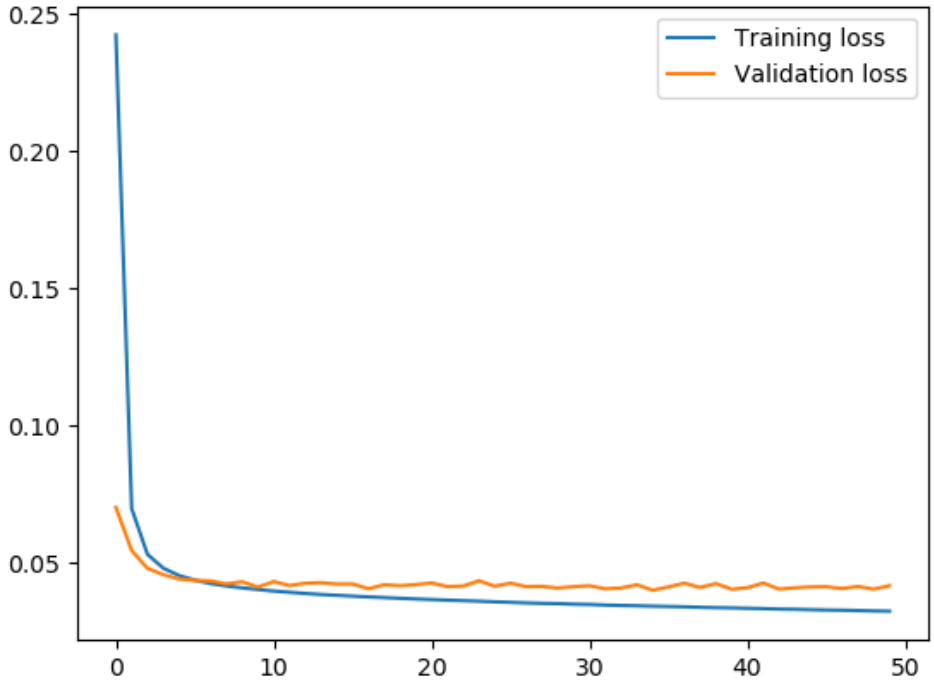


Figure 19 – Validation and training loss against number of epochs

4.4 Final results

As can be seen in Figure 18, the output masks from the U-Net model that is trained on the 1-pixel edge dataset still had significantly thick edges. As such, in the post-processing, the edges are first thinned using the Zhang-Suen thinning algorithm (Zhang & Suen, 1984) and is implemented using the OpenCV library. From there, the line equations of each edge are then obtained using a Probabilistic Hough Transform (Kiryati, Eldar, & Bruckstein, 1991). The output from a sample of self-taken photographs of various indoor environments are shown in Figures 20, 21, 22 and 23.

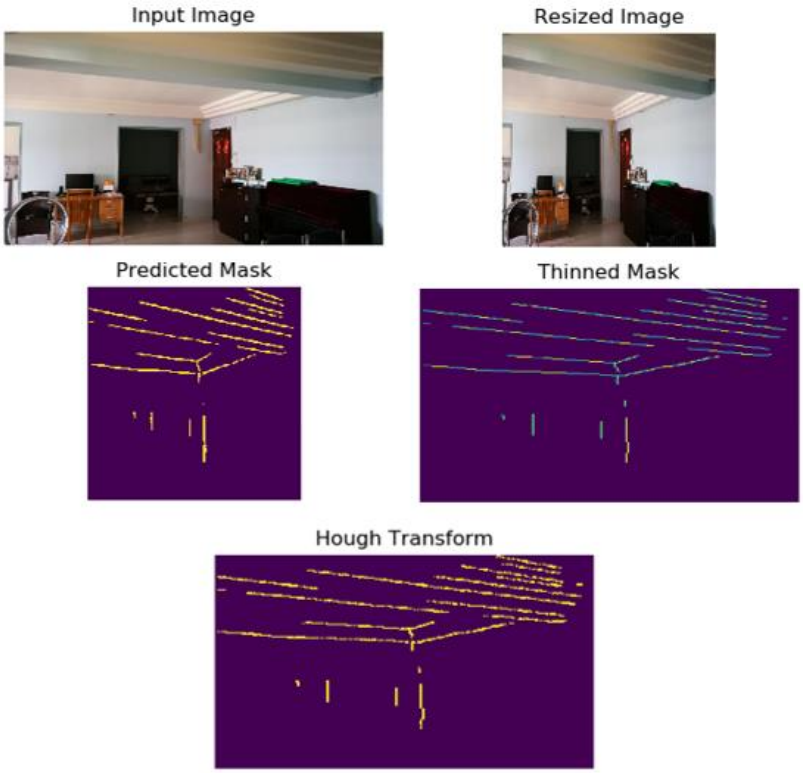


Figure 20 – Final output from image of an apartment

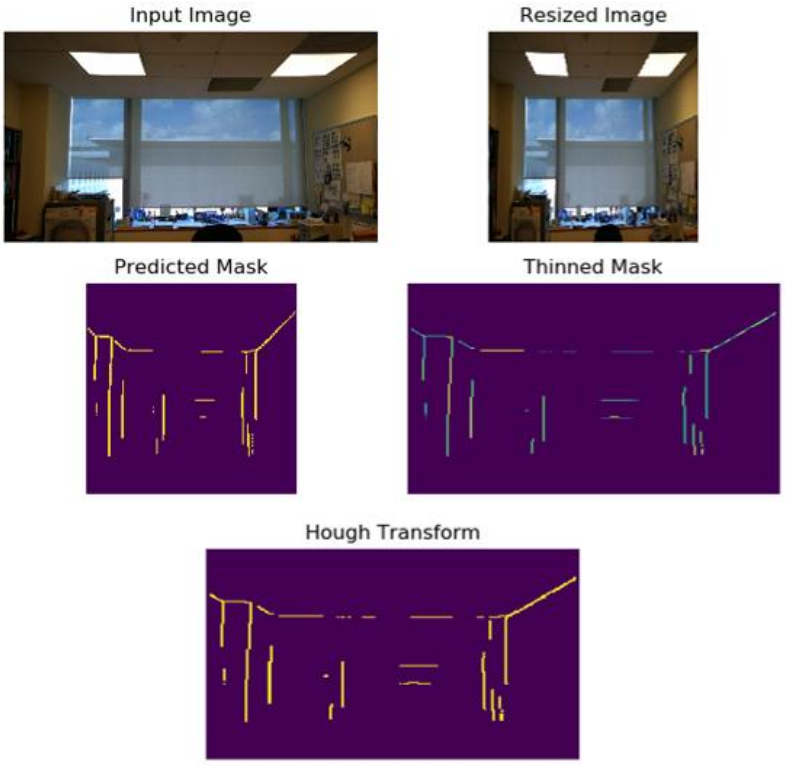


Figure 21 – Final output from image of an office (angle 1)

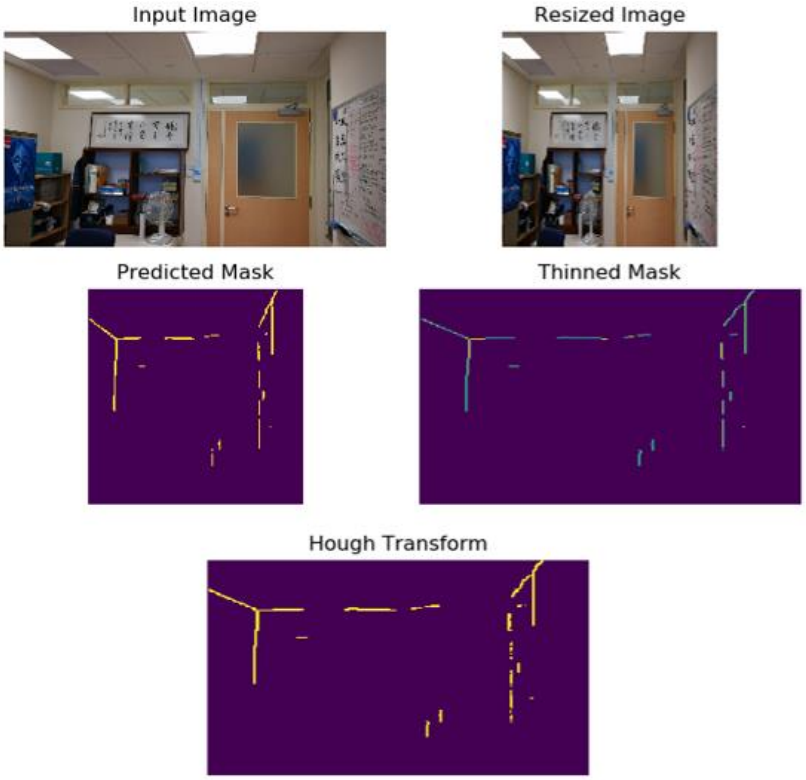


Figure 22 – Final output from image of an office (angle 2)

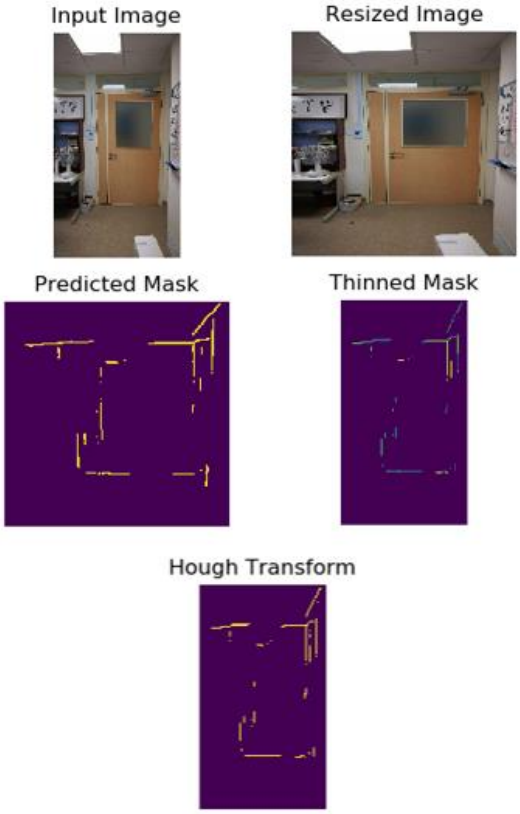


Figure 23 – Final output from image of an office (angle 3)

However, due to the nature of the dataset, there are some scenarios where the model performs exceptionally poorly. In cases where there are curved structural edges, such as arches, or where there are stairs and multiple storeys in the room, the model fails to distinguish the correct edges. Figures 24, 25 and 26 show the output of some of the aforementioned cases.

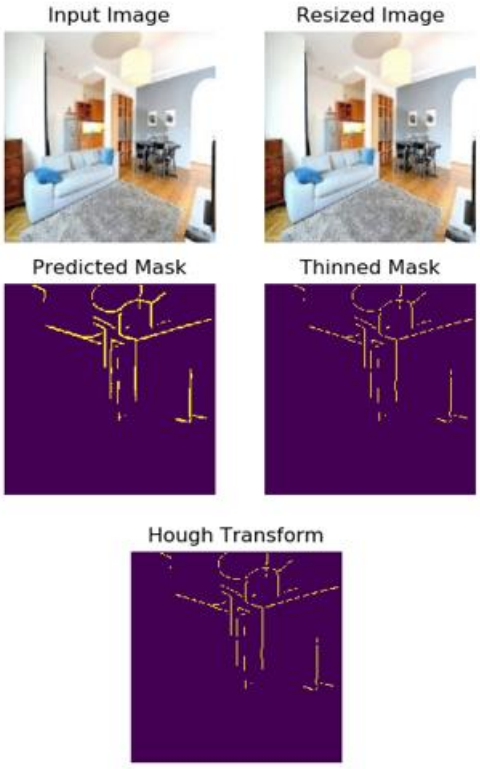


Figure 24 – Final output from image of a room with an arch

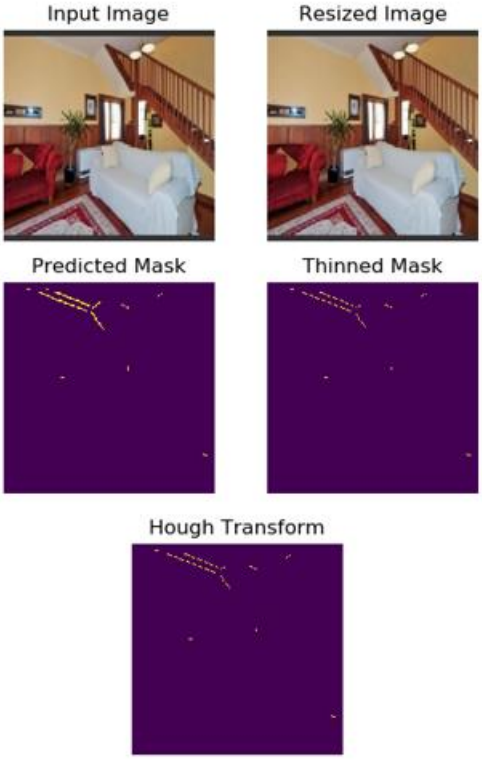


Figure 25 – Final output from image of a room with stairs

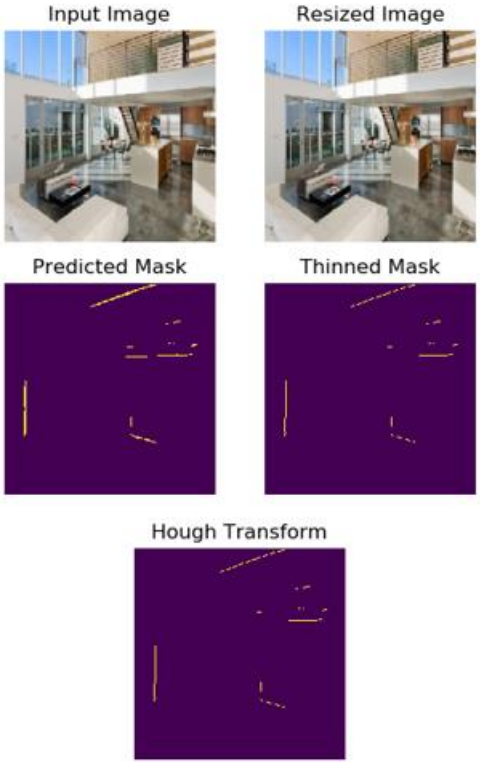


Figure 26 – Final output from image of a room with multiple levels

5. Conclusion

5.1 Summary

This project has created a customised dataset of various labelled rooms and implemented 4 different neural network architectures using PyTorch. These models have been trained, tested and benchmarked against each other to determine that U-Net had the best performance. With further finetuning and refinement, the model based on U-Net is shown to be fairly accurate at identifying structural edges.

These results show that machine learning in the form of FCN is a viable method for detecting structural edges in photographic images and performs better than classical algorithms such as the Canny Edge Detector in terms of discriminating relevant edges. The difference in performance is particularly pronounced in photographs with large amounts of noise or unrelated objects. This project also has shown that U-Net is currently the most promising architecture for this application and has potential to be developed further in the future.

5.2 Future Work

Future work can be done to improve the model by finetuning some of the hyperparameters such as the type of optimiser and the learning rate. There are also some problems with unknown reasons that were encountered during the course of this project that can be explored further, such as the problem with data normalisation and the problem with Cross Entropy loss.

In addition, the current model is not excellent at identifying floor edges since the dataset mostly contains images where the floor edges are hidden by objects. As such, there is a natural bias in the dataset which cannot be corrected easily. To rectify this,

the dataset can be expanded in the future to include more empty rooms so that the model can be trained on a more balanced dataset. One way of doing so is to use a CAD software such as SolidWorks to create digital renders of empty rooms.

Furthermore, the current model also performs poorly at identifying structural edges for rooms that contain curved edges, stairs, and multiple levels. Future work to combat this weakness can be to include more labelled images of these types into the dataset so that the model can be trained to recognise these features. If such images are hard to find, SolidWorks renders may be an alternative as well.

Lastly, some edges in the photographs are partially hidden behind objects, such as furniture and people. Therefore, the visible parts of these edges would be broken into smaller segments. As such, further work needs to be done in developing a robust algorithm that can extrapolate broken line segments to reconstruct the original line such that a wireframe model can be obtained.

References

Amazon. (2018). *Overview: Amazon Mechanical Turk*. Retrieved from Amazon Mechanical Turk: https://www.mturk.com/

B, N. (2017, Sep 11). *Image Data Pre-Processing for Neural Networks*. Retrieved from Becoming Human: https://becominghuman.ai/image-data-pre-processing-for-neural-networks-498289068258

Bushaev, V. (2018, Oct 22). *Adam — latest trends in deep learning optimization.* Retrieved from Towards Data Science: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

Canny, J. (1986, Nov). A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8*(6), 679-698.

Fung, V. (2017, June 16). *An Overview of ResNet and its Variants*. Retrieved from Towards Data Science: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035

Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering, 9*(3), 90-95.

Jain, Y. (2018, Apr 24). *Tensorflow or PyTorch : The force is strong with which one?* Retrieved from Medium: https://medium.com/@UdacityINDIA/tensorflow-or-pytorch-the-force-is-strong-with-which-one-68226bb7dab4

Jonathan Long, E. S. (2014). *Fully Convolutional Networks for Semantic Segmentation.*

Kingma, D. P., & Ba, J. (2014). *Adam: A Method for Stochastic Optimization.* ArXiv.

Koiran, P., & Sontag, E. D. (1998, August 18). Vapnik-Chervonenkis Dimension of Neural Nets. *Discrete Applied Mathematics, 86*(1), 63-79.

Machine Learning Glossary. (2017). *Loss Functions*. Retrieved from Machine Learning Glossary: https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

Marr, B. (2018, October 1). *Forbes*. Retrieved from What Is Deep Learning AI? A Simple Guide With 8 Practical Examples: https://www.forbes.com/sites/bernardmarr/2018/10/01/what-is-deep-learning-ai-a-simple-guide-with-8-practical-examples/#5881c2a08d4b

Olaf Ronneberger, P. F. (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation.*

OpenCV. (2020). *About OpenCV*. Retrieved from OpenCV: https://opencv.org/about/

Pygame. (2020, Mar). *About Pygame*. Retrieved from Pygame: https://www.pygame.org/wiki/about

PyTorch. (2019). *Neural Network Docs*. Retrieved from PyTorch: https://pytorch.org/docs/stable/nn.html

Russell, B. C., Torralba, A., Murphy, K. P., & Freeman, W. T. (2008, May). LabelMe: a database and web-based tool for image annotation. *International Journal of Computer Vision, 77*, 157-173.

Shah, M. (2017, June 1). *Semantic Segmentation using Fully Convolutional Networks over the years*. Retrieved from Github: https://meetshah1995.github.io/semantic-segmentation/deep-learning/pytorch/visdom/2017/06/01/semantic-segmentation-over-the-years.html

Simon Jégou, M. D. (2016). *The One Hundred Layers Tiramisu: Fully Convolutional DenseNets for Semantic Segmentation.*

Sun, C., Shrivastava, A., Singh, S., & Gupta, A. (2017). *Revisiting Unreasonable Effectiveness of Data in Deep Learning Era.* ArXiv.

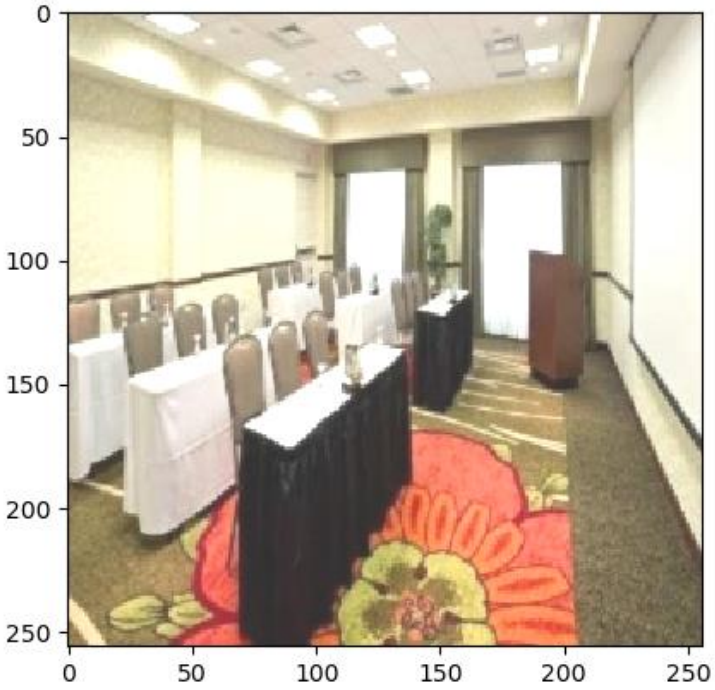
Vijay Badrinarayanan, A. K. (2015). *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation.*

Yu, F., Seff, A., Zhang, Y., Song, S., Funkhouser, T., & Xiao, J. (2015). *LSUN: Construction of a Large-scale Image Dataset using Deep Learning with Humans in the Loop.* Princeton University.

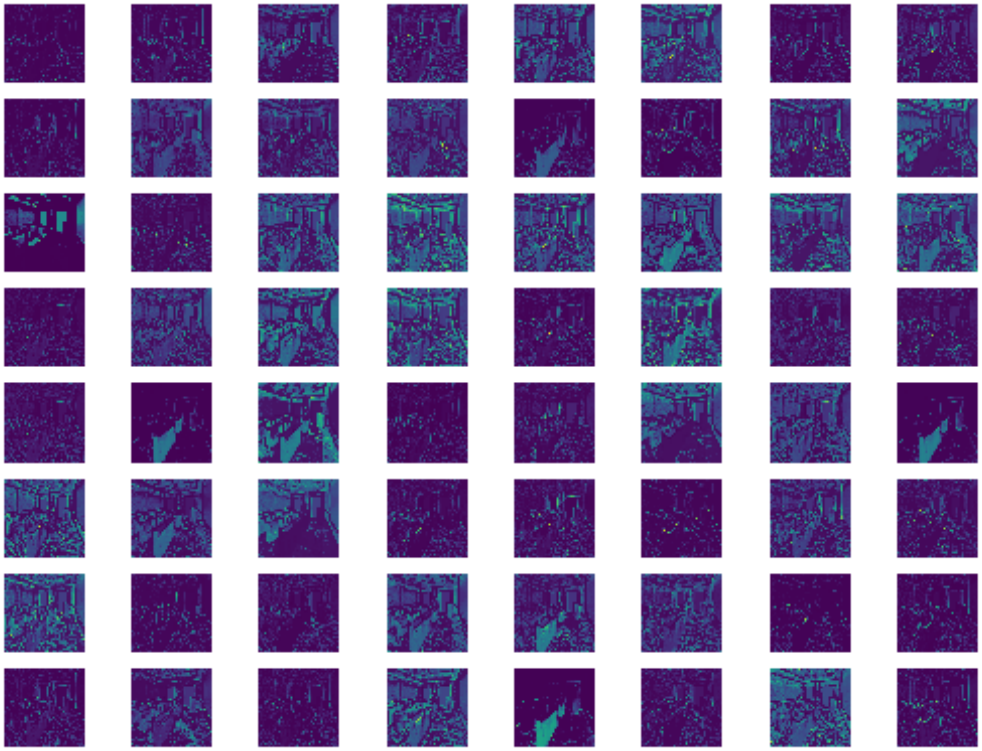
Zhang, T. Y., & Suen, C. Y. (1984). A Fast Parallel Algorithm for Thinning Digital Patterns. *Communications of the ACM, 27*, 236-239.

Appendix A

Input Image

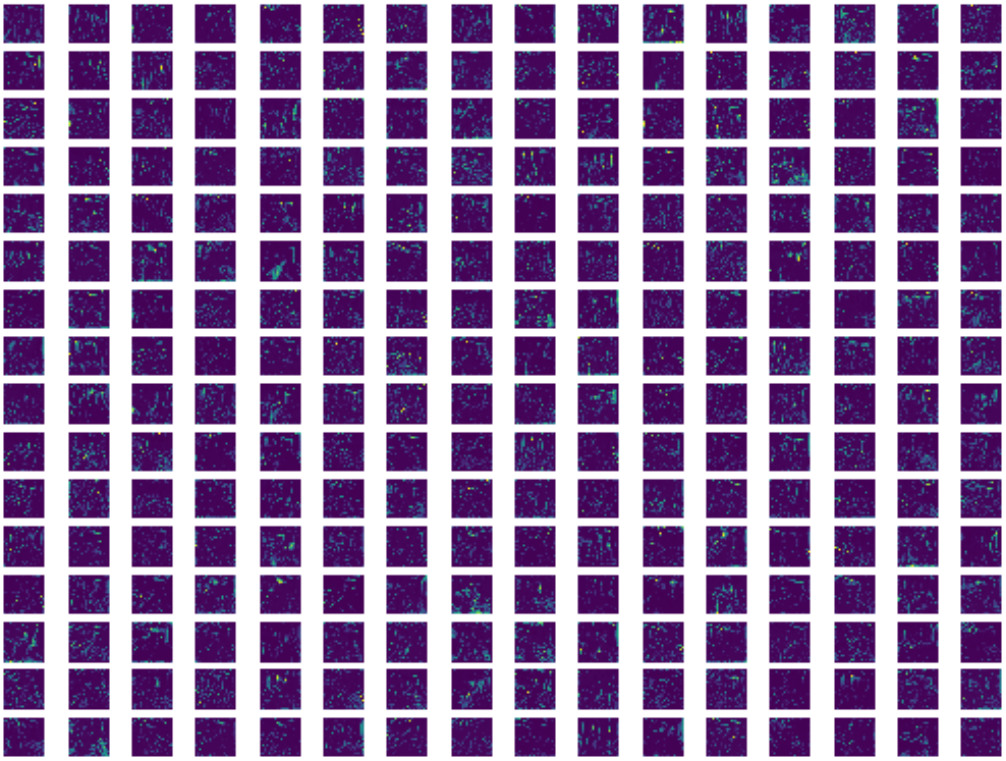


First Layer Convolution Filters



Second Layer Convolution Filters

Third Layer Convolution Filters



Fourth Layer Convolution Filters