COS80029 - Technology Application Project

Project Report

Wound Assessment

Facilitator

Rui Zhou

Project Team

Lucas Qin – 103527269

Hariesh Kumar Ravichandran – 104008697

Andrew Oates – 2473399

Natalie Huang – 103698990

Mona Ma – 103699029

Executive Summary

This report focuses on the application of machine learning to perform complex tasks in wound analysis, including border detection, coin detection and wound measurement, wound assessment, and colour analysis.

It presents two solutions for border detection—one through building a deep learning model and another by extracting wound outlines.

Furthermore, it details methods to detect coins in images for wound measurement, despite background noise. Improving the accuracy of wound assessment through converting pixel measurements to real-world units is also highlighted.

Lastly, it explores the process of colour analysis of wounds, utilising image preparation, colour extraction and categorization, and colour comparison and analysis. The goal of the report is to provide comprehensive insights into the methodologies and technologies used for advanced wound analysis.

Table of contents

1. Introduction **4**
1.1 Overview 4
1.2 Project background 4
1.3 Scope and objectives 4
1.4 Project deliverables 5
2. Border detection **5**
2.1 Solutions Research 5
2.2 Solution 1 Build deep learning model 7
2.3 Solution 2 Extract outline as wound borders 27
3. Coin detection and Wound measurement **32**
3.1 Purpose and method 32
3.2 Pre-process image 33
3.3 Detect all possible coins 37
3.4 Find the correct coin 39
3.5 Output for actual wound area and calculation 46
4 Improving Measurement Accuracy **47**
4.1 Reviewing Existing Measurement Functions 47
4.2 Measuring Objects Extracted from Images 47
4.3 Converting Pixel Measurements to Real-World Units 49
4.4 Final Measurement Functions 50
4.5 Testing Measurement Accuracy 51
4.6 Visualising Results 54
4.7 Saving, Loading and Comparing Results 59
5. Colour analysis **62**
5.1 Solutions Researched 62
5.2 HSV Iteration 63
5.3 Image Preparation 65
5.4 Colour Extraction and Categories 66
5.5 Colour Comparison and Analysis 67
6. Conclusion and future work **68**
7. References **70**
8. Appendix **71**
Team Contribution Breakdown 71

1. Introduction

1.1 Overview

This article is the final report for the wound identification project. It will cover three aspects of the project. Firstly, the article will cover the introduction, content of use, and requirements for the project. Secondly, there will be a discussion and explanation of each team member's technical part, including two methods for extracting the wound's border (by Lucas and Mona), two methods for measuring the actual size of the wound (by Andrew and Natalie) and one method to get the wound's colour information (by Hariesh). Lastly, self-reflection for each team member will be included.

1.2 Project background

This project aims to build a solution that could be used in identifying a patient's wound borders and colour. The project is currently at a very early stage hence our clients are open to all solutions that are feasible. All the wound images will be captured through a screenshot on our client's computer, and in each image, there will be a reference item like a 2-dollar coin to indicate the actual size of the wound. In terms of the wound's border, the client wished to clearly identify the border and extract it; the identified wound area will be further used to compare with reference items to calculate the actual size of the wound. In terms of the wound's colour, the client wished to first identify different colours that existed in a wound and then identify the change of the colour over time by comparing different images. The form of the solution was decided to be a web application at the moment.

1.3 Scope and objectives

The project's objective is to provide a tool for measuring wounds for the MeaSURE Wound Care App. The programme is used to examine colour variations in the same wound over time and quantify the actual wound area using photographs. Finding an appropriate deep learning model and image processing tools to recognise the wound area and colour from photos is the ambition of the project. The aim also involves picking a good reference object, precisely identifying the object, measuring the size of the wound, and scaling the picture to reflect the true size of the wound. Finally, the instrument gives the clinician information on the wound's size and colour percentage for additional treatment procedures.

1.4 Project deliverables

The deliverables of the project include:

- Image processing method to extract wound's border from images;
- Deep learning model to detect wound area from the images;
- Image processing method to detect the reference coin from images;
- Wound measurement method to scale images and get the real size of wounds;
- Colour extract method to get colours from the wound and analysis them

2. Border detection

2.1 Solutions Research

2.1.1 Commercial Machine Learning Services

There are many commercial machine learning tools that are capable of identifying objects and colours, some popular tools include AWS Rekognition, Azure Cognitive, Google Cloud AI platform and IBM Watson. Considering time constraints, only a few services have been researched.

AWS Rekognition is a cloud-based, highly-scalable deep learning technology that could be used together with AWS's S3 service to analyse images or videoS(Mishra, 2019). It provides some ready-to-use functionalities including the detection of labels, faces, personal protective equipment, celebrities, text, inappropriate images, etc. In terms of identification of wound's colour, Rekognition provided detailed colour type, and percentage through image properties in the label detection function, however, Rekognition has a list of labels that it currently could identify, which means Rekognition can only identify very limited items, and the wound is not one of it. Also, AWS Rekognition does not have any ready-to-use function that could extract an object's border directly. Hence, the attention of the research turned to AWS Rekognition's customisation service (custom labels), which allows users to train their own models to identify objects based on their needs. According to AWS's official website (Mishra, 2019), Rekognition provides 3 use cases that outline the possibilities of the service. These use cases include finding brand logos, sports teams' highlight moments and the ripeness of tomatoes in various media. Custom labels greatly extend the objects to be identified, however, Rekognition does not provide customised properties that give access to some of the details of the identified object like border and colour.

Through the research, it is clear that Rekognition has limited use cases, its functionalities focus on the identification of objects, and it is difficult to get binary details of the identified object and proceed for further analysis. Hence, Rekognition is considered to be unsuitable for this project.

Azure Cognitive is a counterpart of AWS Rekognition developed by Microsoft. It is also a cloud-based machine learning service that enables developers to use it in their own applications without relevant AI or machine learning knowledge (Farley, 2023). Cognitive can be divided into five different parts based on different use scenarios, they are Speech, Language, Vision, Decision and OpenAI service. Among these services, Vision service offers functions similar to AWS Rekognition, which are optical character recognition, image analysis, face detection and spatial analysis. These functions are designed to suit different business needs, for example, spatial analysis can be used to detect human presence and movement in video streams. The images analysis function is designed to identify a wide range of features and objects from images(Farley, 2023), for example, it can be used on detecting colour schemes, areas of interest, people, etc. in terms of detection details, Cognitive's Vision provides few access to identified details, for example, in colour scheme detection, Vision could only be used to determine a dominant colour, or whether an image is black&white. It's not possible to get more colour information like type or percentage. In terms of object identification or area of interest detection, Vision uses a bounding box to indicate the captured object or area, the captured object or area then can be further used to count or crop, but it is also not possible to get the border of the object in any form. When it comes to Cognitive's customisation, according to Azure's official website (Farley, 2023), the Image Analysis model customization service does not support users in controlling their own hyper-parameters or exporting trained models to be used in their application without Azure cloud. That indicates that Cognitive's customisation extends existing object type identification but remains the same on accessible details, hence, in consideration of the need of the project, even though Azure Cognitive is capable of dealing with huge size of datasets in a commercial environment, it is not a suitable tool for the project due to the limited level of details it provides for handling.

In conclusion, commercial machine learning services simplify developer's process of implementing machine learning capability to their application and do not require any knowledge of the relevant field to use or train, however, these service has very limited use scenario, and provide the user with a very limited parameter to control and develop, hence, these services are not suitable for this project.

2.1.2 Open source Machine learning library

Apart from commercial machine learning services, there are many open-source machine learning libraries that allow developers to develop their own machine learning model from scratch. Developers are free to use their own image segmentation techniques, architectures, image preparation techniques, etc. And these libraries allow developers to control a large number of parameters to adjust each step of the training and evaluation.

Some of the most popular libraries include TensorFlow, Pytorch, Deeplearning4j, Keras, Onnx, etc.

2.2 Solution 1 Build deep learning model

In the previous study, commercial machine learning services were found to be not suitable for the project, and image processing tools alone also come with their own limitations. Hence, a method of using one of the machine learning frameworks Tensorflow together with Keras, Numpy, OpenCV, and other services and tools to build a total customised model was proposed, which is capable of handling wound border and colour identification, and output border in any preset type like jpg format, text list or png format and so on. The new model will be using Python as the programming language, and even though it relies on Tensorflow and Keras's self-learning capability, it can be integrated with traditional image processing techniques as well, which means that model can be quite flexible in terms of techniques, architectures, and parameters selection to meet specific needs of the project and future extension.

In the current solution, it follows the steps as below.

2.2.1. Assumptions

- All images are taken with a 2 dollar coin as a reference object, and the object is placed in a pre-selected location which is at the same level as the wound and near the wound.
- All images are taken at a good angle.
- All images are taken in good light.
- Wounds take at least 10% of the image.
- Most images only have one wound in one image.
- The deep learning model focuses on one type of wound.

2.2.2 Loading images

There are three functions responsible for loading images and masks, they are ‘load_json_and_get_mask()’, ‘load_images_and_masks_worker()’ and ‘load_image_and_masks()’ functions.

‘load_json_and_get _mask()’ reads the mask's JSON file and uses the requests.get() method to retrieve the mask from the URI provided in the JSON file. It also loads the corresponding image using OpenCV's cv2.imread().

Then, ‘load_images_and_masks_worker()’ either loads the cached image and mask from a .joblib file if it exists or loads the image and mask using the load_json_and_get_mask() function, resizes and pads them using the resize_and_pad_image_and_mask() function, and saves the results to a .joblib file for future use.

Lastly, ‘load_json_and_masks()’ is the main function, it loads all images and their masks from the provided paths. It uses a ThreadPoolExecutor to parallelize the loading across multiple threads and call ‘load_images_and_masks_worker()’, this method increases loading speed significantly. The function also prepares a cache directory to store processed images and masks for faster loading in the future. The cache was introduced as a result of code improvement, before that, loading images took a long time, sometimes loading an image could take 2 to 10 seconds, with cache, the time of loading all images is now less than a second.

2.2.3 Preprocess images

First of all, all images will be resized to a shape of 256x256 pixels, the 256x256 shape is commonly used in resizing datasets. It can be changed to smaller or bigger to adapt to the project, bigger size means more pixels for analysing, hence could sometimes require extra computational resources. Considering that most dataset images are not square, each image will need to add appropriate paddings after resizing to become a 256x256 shape. Fig.1 defines which side to resize to 256 pixels. If the image's width is greater than its height, the image’s width will be set to 256 pixels, and its height will be set according to the original image’s width/height ratio. In this way, the image's ratio can be kept as original.

```python
def resize_with_aspect_ratio(image, target_size):
    h, w = image.shape[:2]
    aspect_ratio = w / h

    if w > h:
        new_width = target_size
        new_height = int(new_width / aspect_ratio)
    else:
        new_height = target_size
        new_width = int(new_height * aspect_ratio)

    return cv2.resize(image, (new_width, new_height))
```

Figure 1: Resize images to keep the original width/height ratio

In order to make a normal image become a dataset to train the model, each image needs to have a corresponding mask. A mask is an image that outlines the area of interest as white colour, and other areas as black colour, it helps the model to learn and identify the important part in an image. A mask can be generated by using tools like OpenCV, Labelbox, etc, all images will need manual annotation on these tools and that means a large amount of repeated work. Fig.2 shows the effect of a mask applied in original images using Labelbox.

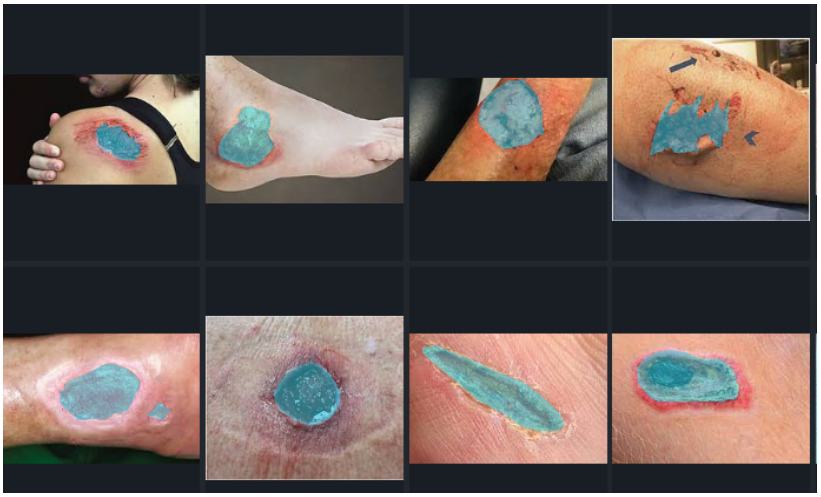

Figure 2: Effect of applying the mask to an image

Figure 3: Example of a mask

Also note that a mask needs to be resized exactly as its original image. Then use OpenCV's cv2 library to extract each image's colour image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB). Fig.3 shows how the machine reads an image, it annotates each pixel with a number, and each number represents a colour. The above cv2 uses an RGB method to represent a colour, which means each colour will be further divided into blue, green and red (Venkatesh, 2021). As for the mask, since it is black&white, it can be converted to binary data using the Numpy library directly as mask = np.array(Image.open(BytesIO(response.content)).convert('L')).

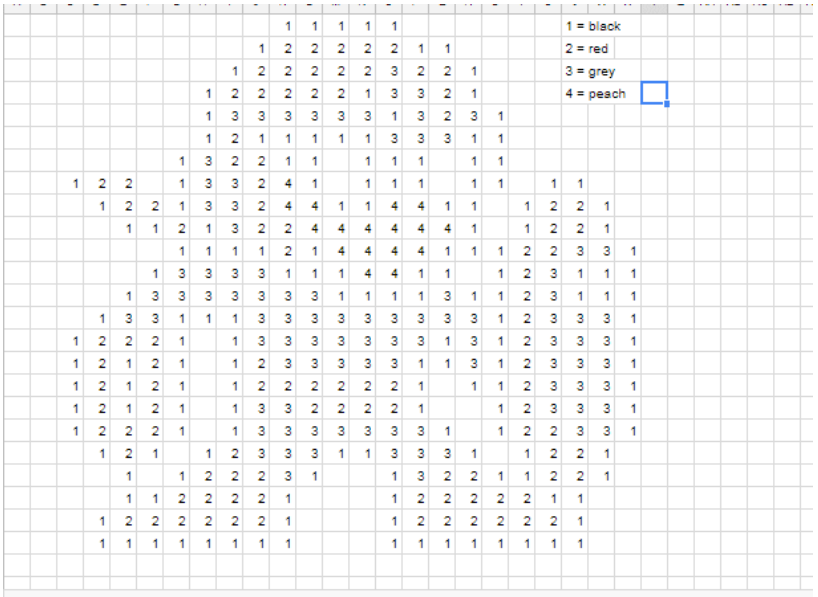

Figure 4: How the machine read an image (Venkatesh, 2021)

2.2.4 Image augmentation

Image augmentation is a process between image loading and model training. It increases the diversity of data available for training and increases the quantity of training datasets by transforming the images into different versions by rotating, scaling, flipping and shifting. It makes the best use of a limited training dataset and improves the performance of the model in terms of reducing overfitting and increasing robustness.

In Fig.5, there's a basic version of image augmentation by Lucas and was further developed by Mona in the latest code version. The code started creating a dictionary that contains all the parameters needed for data augmentation, which include rotation, shifting, shear intensity, zooming, brightness, and flipping. The specific value of these parameters was adjusted carefully to avoid overfitting based on experience. Then called the 'ImageDataGenerator' from Keras library to apply augmentation on wound images and masks separately to ensure each image could have the same augmentation with its corresponding mask.

```
# Data augmentation
data_gen_args = dict(rotation_range=20,             # Increase rotation range
                     width_shift_range=0.1,         # Increase width shift range
                     height_shift_range=0.1,        # Increase height shift range
                     shear_range=0.1,               # Increase shear range
                     zoom_range=0.1,                # Increase zoom range
                     brightness_range=(0.9, 1.1),
                     horizontal_flip=True,
                     vertical_flip=True,            # Add vertical flipping
                     fill_mode='nearest')
image_datagen = ImageDataGenerator(**data_gen_args)
mask_datagen = ImageDataGenerator(**data_gen_args)

image_datagen.fit(X_train, augment=True, seed=42)
mask_datagen.fit(y_train, augment=True, seed=42)
```

Figure 5: Basic image augmentation

Figure 6 provides the final version of image augmentation, it includes the following steps:

- Loop through images and masks: The code goes through each image and corresponding mask from the given batches.
- Set up a series of transformations: The code then establishes a sequence of image transformations. These transformations include flipping the image horizontally and vertically, rotating the image, adjusting the brightness and contrast, shifting and scaling the image, changing the gamma settings (to simulate different illumination conditions), blurring, adding noise, and distorting the image.
- Apply the transformations: Each transformation is applied to both the image and the mask, with a certain probability (represented by 'p') for each transformation. This ensures a variety of transformations are applied, but not all at once. It's like shuffling a deck of cards, with some cards (transformations) being picked more often than others.
- Save the transformed images and masks: After the transformations are applied, the transformed or 'augmented' image and mask are then stored separately for later use. This new, augmented dataset is used to train our machine learning model, helping it learn to generalise from varied data and perform better on unseen data.

In the bigger picture, these transformations help make the machine learning model more robust. The model becomes better at handling real-world variations in data as it has seen similar variations during training.

```python
def augment_data(images, masks, batch_size, image_datagen, mask_datagen):
    while True:
        idx = np.random.permutation(images.shape[0])
        images = images[idx]
        masks = masks[idx]

        for i in range(0, len(images), batch_size):
            batch_images = images[i:i+batch_size]
            batch_masks = masks[i:i+batch_size]

            # Apply image_datagen and mask_datagen augmentations
            batch_images = image_datagen.flow(batch_images, batch_size=batch_size, seed=42).next()
            batch_masks = mask_datagen.flow(batch_masks, batch_size=batch_size, seed=42).next()

            aug_images = []
            aug_masks = []

            for img, mask in zip(batch_images, batch_masks):
                # Apply Albumentations library augmentations
                augmented = Compose([
                    ElasticTransform(alpha=1, sigma=50, alpha_affine=50, p=0.5),
                    A.HorizontalFlip(p=0.5),
                    A.VerticalFlip(p=0.5),
                    A.RandomRotate90(p=0.5),
                    A.RandomBrightnessContrast(p=0.5),
                    A.ShiftScaleRotate(shift_limit=0.1, scale_limit=0.1, rotate_limit=20, p=0.5),
                    A.RandomContrast(limit=0.2, p=0.5),
                    A.RandomGamma(gamma_limit=(80, 120), p=0.5),
                    A.RandomCrop(height=128, width=128, p=0.5),
                    A.GaussianBlur(blur_limit=3, p=0.5),
                    A. GaussNoise(var_limit=(10, 50), p=0.5),
                    A.OpticalDistortion(distort_limit=0.05, shift_limit=0.05, p=0.5)
                ])(image=img, mask=mask)

                aug_images.append(augmented['image'])
                aug_masks.append(augmented['mask'])


            yield np.array(aug_images), np.array(aug_masks)
```

Figure 6: Further image augmentation

2.2.5 Building model with U-net

There are many options when choosing an architecture to train a model, it can also be DeepLab, Mask_RCNN or others. However, U-Net is my primary choice due to the following reasons.

Firstly, U-Net is widely used for biomedical image segmentation, it performs well pixel-wide to identify areas of disease or other medical-related features of interest.

Secondly, U-Net is versatile and flexible in working with other techniques like data augmentation, custom loss function, and other image processing techniques. Considering that not all tasks of other team members use machine learning, for instance, colour detection can be done by using image processing techniques, then it is important that U-Net can be compatible with these techniques.

Thirdly, U-Net is well known for its good performance with small datasets, considering that it is difficult for collecting large amounts of real wound images as training data due to privacy policies, especially wounds of a particular type, hence U-Net seems to be the perfect architecture to me.

Lastly, U-Net can capture objects of interest that have various scales due to its expansive architecture.

SegNet and DeepLabV3+ also have been built and tested. DeepLabV3+ is generally more powerful than SegNet, however, it requires a large amount of dataset and more computational resources than U-Net and SegNet, this will also lead to more training and configuration time. Considering we are unable to obtain a large amount of dataset, DeepLabV3+ was not the perfect solution for this project, in fact, I tested a basic version of DeepLabv3+, and its performance was similar to U-Net under current circumstances while with more complex codes. As for SegNet, it seems like my Windows system has an issue in installing a critical component, ‘h5py’ to run SegNet, hence I was not able to run it successfully, it could be an alternative architecture in the future to try since it does not require a large amount of dataset as well.

In current U-Net architecture, it first specifies the input format of training images to be the size of 256 * 256 pixels with 4 channels (red, green, blue, and alpha channel), the first 3 channels represent the colour components of an image, and the fourth channel was designed to deal with overlaying images as it can preserve transparency information and extra information of the image like colour space, it could benefit those tasks that involve .png format’s training data.

Then secondly, there are 4 blocks of layers as encoders and 3 blocks of layers as decoders. On the one hand, each encoder block consists of a 2D convolution layer (Conv2D), a batch normalisation layer (BatchNormalization), and a 2D max pooling layer (MaxPooling2D). The number of filters in the Conv2D layer doubles with each subsequent block, starting from 26 and ending with 205. The Conv2D layers use a 3x3 kernel and 'relu' as the activation function. Padding is set to 'same' to maintain the input's height and width. The MaxPooling2D layers have a pool size of (2, 2), meaning they reduce the spatial dimensions (height and width) of the feature maps by half. On the other hand, each block of the decoder begins with an upsampling layer (UpSampling2D) followed by concatenation (Concatenate) with the feature map from the corresponding block in the encoder part. This is followed by a Conv2D layer and a BatchNormalization layer. The number of filters in the Conv2D layer decreases with each subsequent block, starting from 102 and ending with 26.

In the end, there's a Conv2D output layer with one filter and a sigmoid activation function. The sigmoid activation function was used to define output values to be between 0 and 1 and make it suitable for binary segmentation tasks, for example generating binary masks as the result of the model's learning outcome.

Figure 7 and Figure 8 are the current and original U-Net structures. Compared to the original version of U-Net, first of all, I added an additional channel to store additional information about training images. Then the code reduced the complexity of the original U-Net to reduce training time, including reducing the convolutional layers number and filter number. By adjusting the number of filters (reducing the model's depth), the U-Net structure successfully avoids overfitting on the currently provided dataset. In addition, I added batch normalisation layers after each Conv2D layer to help accelerate learning, reducing overfitting and making the model less sensitive to initial starting weights.

```
# Build the model (U-Net)
def build_unet(input_shape=(256, 256, 4)):
    inputs = tf.keras.Input(input_shape)

    conv1 = Conv2D(26, (3, 3), activation='relu', padding='same')(inputs)
    bn1 = BatchNormalization()(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(bn1)

    conv2 = Conv2D(52, (3, 3), activation='relu', padding='same')(pool1)
    bn2 = BatchNormalization()(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(bn2)

    conv3 = Conv2D(102, (3, 3), activation='relu', padding='same')(pool2)
    bn3 = BatchNormalization()(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(bn3)

    conv4 = Conv2D(205, (3, 3), activation='relu', padding='same')(pool3)
    bn4 = BatchNormalization()(conv4)

    up7 = Concatenate()([UpSampling2D(size=(2, 2))(bn4), conv3])
    conv7 = Conv2D(102, (3, 3), activation='relu', padding='same')(up7)
    bn7 = BatchNormalization()(conv7)

    up8 = Concatenate()([UpSampling2D(size=(2, 2))(bn7), conv2])
    conv8 = Conv2D(52, (3, 3), activation='relu', padding='same')(up8)
    bn8 = BatchNormalization()(conv8)

    up9 = Concatenate()([UpSampling2D(size=(2, 2))(bn8), conv1])
    conv9 = Conv2D(26, (3, 3), activation='relu', padding='same')(up9)
    bn9 = BatchNormalization()(conv9)

    output = Conv2D(1, (1, 1), activation='sigmoid')(bn9)

    return tf.keras.Model(inputs=inputs, outputs=output)
```

Figure 7: Current U-Net architecture

```python
# Build the model (U-Net)
def build_unet(input_shape=(128, 128, 3)):
    inputs = tf.keras.Input(input_shape)

    conv1 = Conv2D(64, (3, 3), activation='relu', padding='same')(inputs)
    conv1 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(128, (3, 3), activation='relu', padding='same')(pool1)
    conv2 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Conv2D(256, (3, 3), activation='relu', padding='same')(pool2)
    conv3 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

    conv4 = Conv2D(512, (3, 3), activation='relu', padding='same')(pool3)
    conv4 = Conv2D(512, (3, 3), activation='relu', padding='same')(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)

    conv5 = Conv2D(1024, (3, 3), activation='relu', padding='same')(pool4)
    conv5 = Conv2D(1024, (3, 3), activation='relu', padding='same')(conv5)

    up6 = Concatenate()([UpSampling2D(size=(2, 2))(conv5), conv4])
    conv6 = Conv2D(512, (3, 3), activation='relu', padding='same')(up6)
    conv6 = Conv2D(512, (3, 3), activation='relu', padding='same')(conv6)

    up7 = Concatenate()([UpSampling2D(size=(2, 2))(conv6), conv3])
    conv7 = Conv2D(256, (3, 3), activation='relu', padding='same')(up7)
    conv7 = Conv2D(256, (3, 3), activation='relu', padding='same')(conv7)

    up8 = Concatenate()([UpSampling2D(size=(2, 2))(conv7), conv2])
    conv8 = Conv2D(128, (3, 3), activation='relu', padding='same')(up8)
    conv8 = Conv2D(128, (3, 3), activation='relu', padding='same')(conv8)

    up9 = Concatenate()([UpSampling2D(size=(2, 2))(conv8), conv1])
    conv9 = Conv2D(64, (3, 3), activation='relu', padding='same')(up9)
    conv9 = Conv2D(64, (3, 3), activation='relu', padding='same')(conv9)

    output = Conv2D(1, (1, 1), activation='sigmoid')(conv9)

    return tf.keras.Model(inputs=inputs, outputs=output)
```

Figure 8: Original U-Net architecture

2.2.6 Train the model

First of all when training the model, the code compiled the model with Adam optimizer, the learning rate is set to 0.0005 while the default value is 0.001. The learning rate is one of the most important parameters when adjusting the performance of the model. If the learning rate is too big, it can lead to unstable training and cause the loss function to diverge or oscillate. The model may fail to converge and have difficulty finding the optimal solution.

On the other hand, if the learning rate is too small, the training process may become very slow. The model will take a long time to get the result. Generally, the learning rate will need to match with ‘epochs’ number, epochs number is another critical parameter that needs to adjust based on empirical judgement. An epoch is the iteration of the model train on the whole training dataset. When the epoch increases, the learning rate should decrease, the specific combination of the values will rely on manually testing the performance of the model output. In this project, when the learning rate is 0.001, epochs should be around 5, when the learning rate is 0.0005, epochs should be around 8. In fact, epochs also change with the dataset, when adding similar images to the current dataset, the epochs number may need to reduce, otherwise, if adding totally different types of images with distinguished characters, epochs may need to increase to better learn the new features bring by new images.

Secondly, call the image augmentation function mentioned above to enhance the training dataset. The train_generator and val_generator are generators that use the ImageDataGenerator objects to generate augmented batches of data during the training process. These generators then output batches of augmented images and their corresponding masks for training and validation. These images and their corresponding masks are then provided to the model through ‘fit()’ function. In the ‘fit()’ function, except for epochs, there’s another important parameter ‘batch_size’ that needs to be adjusted based on experience. Batch size defines how much data to feed to a model at a time, batch size can have a significant impact on training performance. On the one hand, when the batch size is too large, the model could potentially ignore some image features, at the same time, the training would take more memory resources and time. On the other hand, if the batch size is too small, it will increase the risk of overfitting the training data and also cause fluctuations in the model’s training and performance. Hence it is important to find a balance value of batch size. Currently using a batch size of 8 generates a good result. In the end, save the trained model.

In addition, an ‘EarlyStopping’ function from Keras library was used to stop the training process when validation loss does not improve for 5 consecutive epochs, it helps to prevent overfitting and improve the efficiency of training.

```
model = build_unet()

# Compile the model, learning rate default is 0.001
optimizer = Adam(learning_rate = 0.0005)
model.compile(optimizer=optimizer, loss=BinaryCrossentropy(), metrics=[dice_coefficient])


# Train the model

train_generator = augment_data(X_train, y_train, batch_size, image_datagen, mask_datagen)
val_generator = augment_data(X_val, y_val, batch_size, image_datagen, mask_datagen)

early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1, restore_best_weights=True)

history = model.fit(train_generator, steps_per_epoch=len(X_train) // batch_size, validation_data=val_generator,
                    validation_steps=len(X_val) // batch_size, epochs=8)

# Save the trained model
model.save(model_path)
```

Figure 9: Train the model with suitable parameters

2.2.7 Fine-tuning the model

Fine-tuning a model is an essential part of a machine learning project, especially for the current task. There are several reasons why the model needs fine-tuning:

- Adapting to our task: Even though our model was initially trained on a large dataset, the tasks it was originally trained for may differ from ours. To ensure our model performs well on our specific task, we need to make some adjustments. This process is what we call fine-tuning.
- Improving model performance: Fine-tuning allows us to boost our model's performance. The initial model might not have delivered satisfactory results on our task. By fine-tuning, it was able to adjust the model to better understand our data and consequently enhance its performance.
- Handling limited data: The project might not have a vast amount of data. But we could leverage a model pre-trained on a large dataset to understand general patterns and trends. By fine-tuning this model on our smaller dataset, we can make it more relevant and efficient for our specific task.
- Preventing overfitting: Overfitting happens when a model learns too much from the training data to the point where it performs poorly on new, unseen data. Fine-tuning helps us strike a balance between learning from the data and maintaining the model's ability to generalise to new data.
- Efficient use of resources: Training a model from scratch requires a lot of computational power and time. By using a pre-trained model and fine-tuning it, we can save on these resources, as the model already comes with learned patterns and trends from its original training.

Therefore, fine-tuning our model was necessary to adapt it to our specific task, improve its performance, efficiently utilise our data and computational resources, and ensure it can effectively handle new, unseen data.

We built two small fine-tuning datasets. One dataset contains 65 images in which the two-dollar coin is near the wound border, the other dataset contains around 80 images with only a single type of real wound.

Then, we built a fine-tuning method to fine-tune our U-net model. The steps of getting a fine-tuning model are the same as the last training model section, and the code is displayed below.

```
def fine_tune_model(model, new_images_json_path, new_masks_json_path, original_images, resized_binary_masks):
    batch_size = 8
    # Make some layers trainable, for example, the last 5 layers
    for layer in model.layers[-5:]:
        layer.trainable = True

    # Compile the model with a potentially different learning rate
    optimizer = Adam(learning_rate=0.0001)
    model.compile(optimizer=optimizer, loss=BinaryCrossentropy(), metrics=[dice_coefficient])

    # Load the new dataset images and masks
    X_new, y_new = load_images_and_masks(new_images_json_path, new_masks_json_path)

    X_train_new, X_val_new, y_train_new, y_val_new = train_test_split(X_new, y_new, test_size=0.2, random_state=42)

    image_datagen_new = ImageDataGenerator(**data_gen_args)
    mask_datagen_new = ImageDataGenerator(**data_gen_args)

    image_datagen_new.fit(X_train_new, augment=True, seed=42)
    mask_datagen_new.fit(y_train_new, augment=True, seed=42)

    train_generator_new = zip(image_datagen_new.flow(X_train_new, batch_size=batch_size, seed=42),
                              mask_datagen_new.flow(y_train_new, batch_size=batch_size, seed=42))

    val_generator_new = zip(image_datagen_new.flow(X_val_new, batch_size=batch_size, seed=42),
                            mask_datagen_new.flow(y_val_new, batch_size=batch_size, seed=42))

    history = model.fit(train_generator_new, steps_per_epoch=len(X_train_new) // batch_size, validation_data=val_generator_new,
                        validation_steps=len(X_val_new) // batch_size, epochs=5)

    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.title('Training and Validation Loss')

    plt.show()

    for i, (original_image, resized_mask) in enumerate(zip(original_images, resized_binary_masks)):
        display_image = convert_image_for_display(original_image)
        cv2.imshow(f'Original Image {i}', display_image)
        cv2.imshow(f'Resized Binary Mask {i}', resized_mask)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

    # Save the fine-tuned model
    timestamp = int(time.time())
    model.save(f'models/model_finetuned_{timestamp}.h5')
    return model
```

Figure 10: Fine-tuning model function

We got a really good result after fine-tuning the model, and the masks of wounds from the results of the model are more accurate. The following figure displays the comparison of the result from our first U-net model and the fine-tuning model.

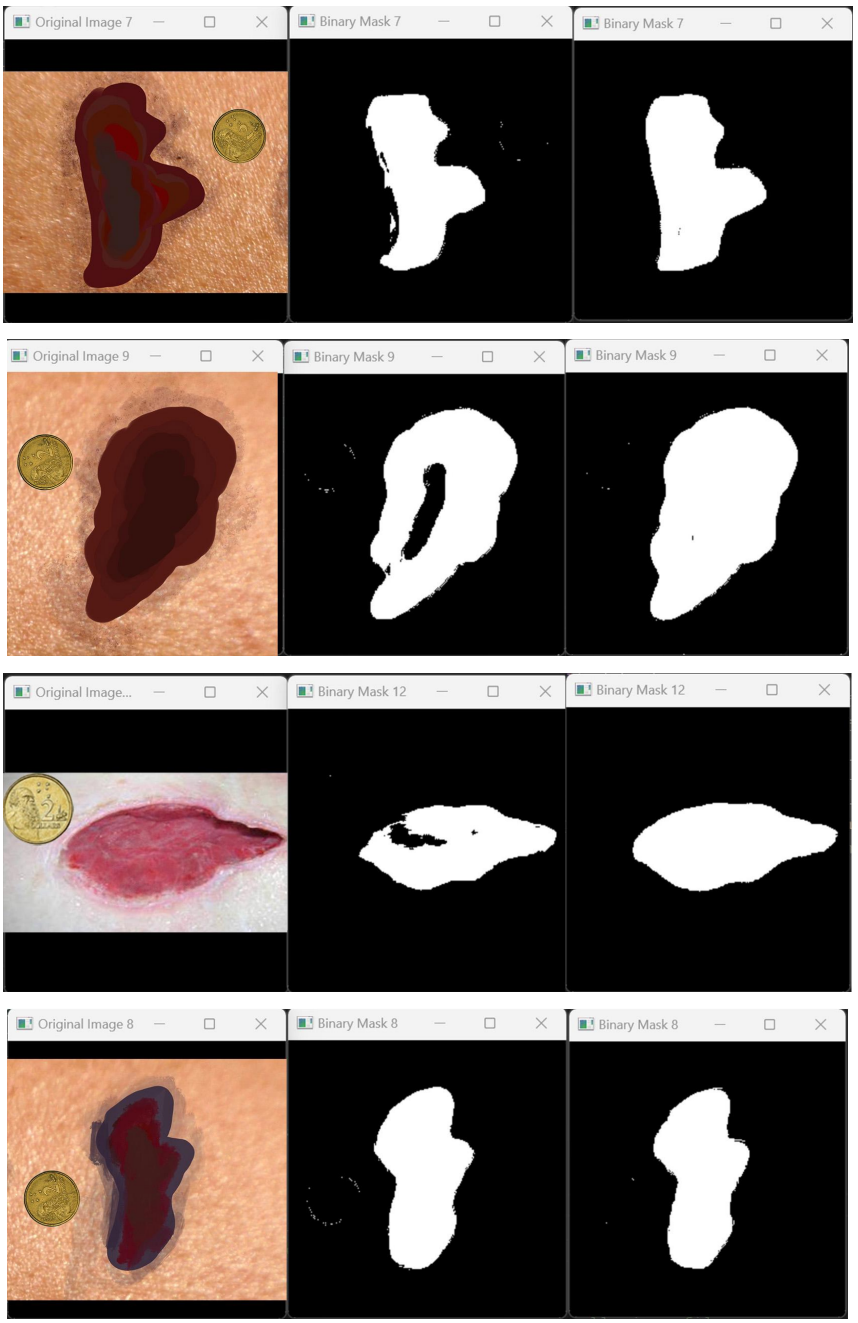

Figure 11: Comparison of U-net model and fine-tuning model

2.2.8 Evaluate the model

To evaluate the model, first of all, there's a folder storing all images that need to be evaluated as shown in Fig.9. There are three types of images for evaluation, they are corresponding to 3 types of training images, type 1 is the basic artificial wound images without any noise on the background, type 2 is the artificial wounds with real skin texture on the background as added noise, type 3 is the real wound that is similar to artificial wound. The load_fake_evaluation_images() function uses OpenCV to load and read evaluation images from the 'fake_evaluation' folder, then converts the default BGR colour space used by OpenCV to RGB in order to display the correct colour of the image.

Secondly, the model uses the predict() method to predict masks for the evaluation images with a set threshold value, which is used to control the edge of the wound and convert the predicted masks into binary masks.

Thirdly, after getting the binary mask, a function called 'extract_wound_area()' was defined to get the accurate coordinates of the wound area as shown in Fig.10.

Finally, the extracted wound area as a Numpy array can be provided to other team members for calculating the wound's actual size and colour information. Using OpenCV's 'imshow' method, I can display evaluation and their evaluation result side by side, it allows me to check the learning result of the machine learning model and adjust the training process, evaluation results can be seen in Fig.11. The images with colour are original images, and the black and white images are binary masks generated by the model through training and learning. The result shows the model can deal with artificial wounds and some kinds of real wounds really well. It verified that the training data we are using, the image processing and augmentation, the model's architecture and evaluation thresholding control were a success. The method that uses artificial wounds to simulate real wounds for training was also proven to be a success. However, there are many types of wound and skin conditions out there, and due to the constraint of the project's time and resources, I am unable to continue with other types of wound identification. In the future, if the project is to be continued, it is suggested that in order to identify a particular type of wound, first provide this particular type of real wound images of at least 500 and make sure the image to not include other irrelevant items like another hand, pen, etc. Otherwise, more datasets needed to be provided to help the model to exclude irrelevant items.

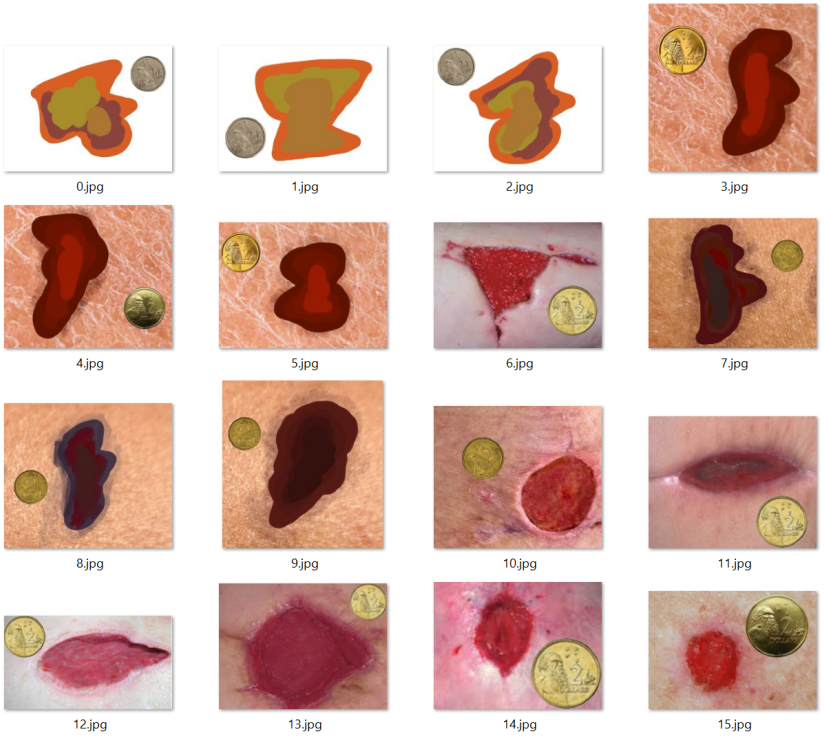

Figure 12: Evaluation dataset

```python
# Get wound area from evaluation dataset
def extract_wound_area(image, binary_mask):
    # Apply morphological dilation to the predicted mask
    kernel = np.ones((5, 5), np.uint8)
    dilated_mask = cv2.dilate(binary_mask, kernel, iterations=1)

    # Find contours in the dilated mask
    contours = measure.find_contours(dilated_mask, 0.5)

    # Find the largest contour
    max_contour = max(contours, key=lambda x: len(x))

    # Create an array of tuples containing the x and y coordinates of each point in the contour
    wound_area = np.array([(int(point[1]), int(point[0])) for point in max_contour])

    return wound_area
```

Figure 13: extract_wound_area() function

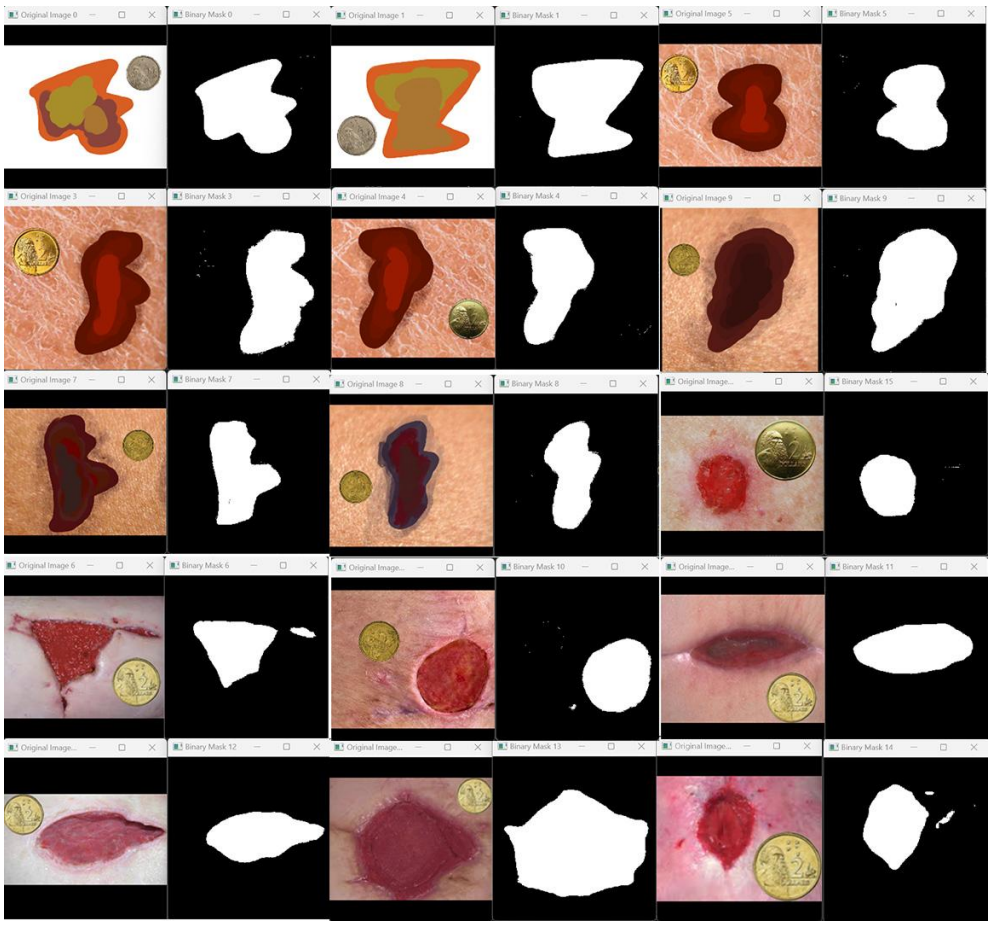

Figure 14: Model's output result

2.2.9 Making artificial dataset

2.2.9.1 Rationale of using artificial wound images

In order to train a model to be able to identify various wound borders, it is important to collect sufficient real wound images for training. There could be multiple ways to collect wound images.

1. Provided by the client.
2. Buy from commercial image databases.
3. Search on the internet to find free wound images or databases.

First of all, our client does not have a sufficient amount of wound images that could be provided for this project since wound images need approval from patients.

Secondly, commercial image databases could be costly, for example, shutterstock.com costs 189 USD a month and allows the download of up to 50 images (Stock photo, royalty-free image prices and plans 2023).

Considering that most of the wound's images are for commercial purposes and a large number of images are needed, a commercial images database is not the best solution as well. Thirdly, there are a limited amount of wound images in Google's search results due to multiple Google policies on medical content restriction, for example, Google Images' Policies (Google search policies for Images & Video Boxes 2023). In the end, only a total of around 130 wound images were collected and transformed into training data with corresponding masks (stored in project folder 'real_wound' and 'real_wound_masks'), but far from enough. Some medical databases are free to the public, for instance, Medetee Wound Database and Chronic Wounds Image Database. Medetee Wound Database focus on complex and rare cases while Chronic Wounds Image Database was photos taken by depth camera for 3D scanning, and is not applicable to this project. Some other medical databases like UpToDate and ClinicalKey have strict access.

Considering the limited development time for this project, it is better to start building the machine learning model as soon as possible. Hence, the best option was to create artificial images that are similar to a certain type of real wound, and test it to assess and verify the rationality and feasibility of the model. Once the model design and methods have been proven to be a success, it would work for other types of wounds as long as training with the particular type of wound images.

2.2.9.2 Types of wound images

As shown in Fig.12, there are a few types of images that have been added to training data gradually over the project development.

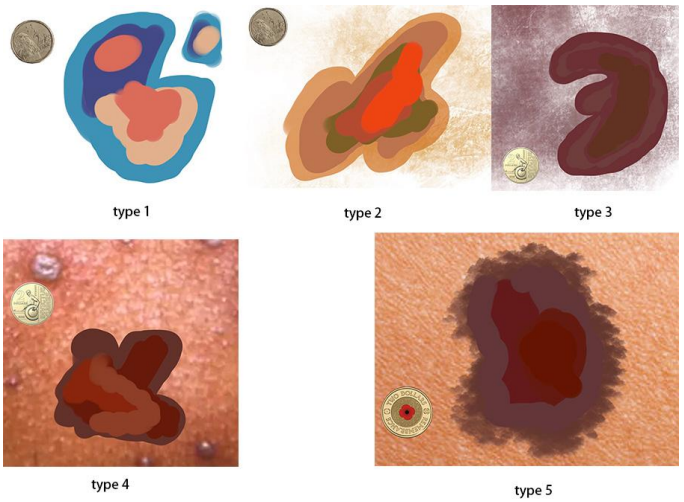

Figure 15: Different artificial training images

At the beginning, around 65 type 1 images were added as training datasets. During the training process, training loss and validation loss are commonly used to monitor the model's learning performance from the training data. In Fig.13, the training and validation loss curve shows that both curves fluctuated from epoch 1 to 14, which means the model could not extract the key features from the provided images yet, ideally, the training and validation loss will be at their highest point from the beginning and decrease gradually until they reach a point when the curve becomes flatten, that means the model is able to adjust learning from the data effectively. If at a point, the validation increases again after a long journey of decrease, that means the model is starting to overfit from the epochs on x-axis; generally, the lower the loss values, the better performance the model can achieve. Fig.13's situation could possibly be caused by an insufficient amount of dataset, hence we added type 2 and type 3 images of a total of around 65, which means we doubled the size of the original dataset.

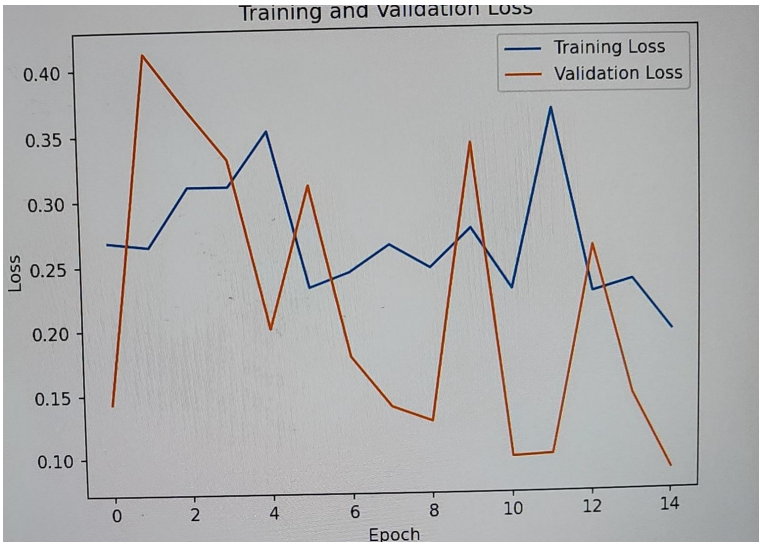

Figure 16: Training and validation loss curve (after adding type 1 images)

Type 2 typically added noise to the background, and type 3 not only added noise with a similar colour to the wound but also changed the appearance of coins, this would further help the model to learn the border feature from training data, it also helps the model to recognise and ignore coins when generating wound border masks. After adding type 2 and 3 images, we can see from Fig.14 that the model became less fluctuated, that's a good sign, meaning the model now narrows down key features of these images, but it is still struggling in getting the feature we expect, the border. We assumed that was because the newly added images came with new different features like new coins and backgrounds full of noise.

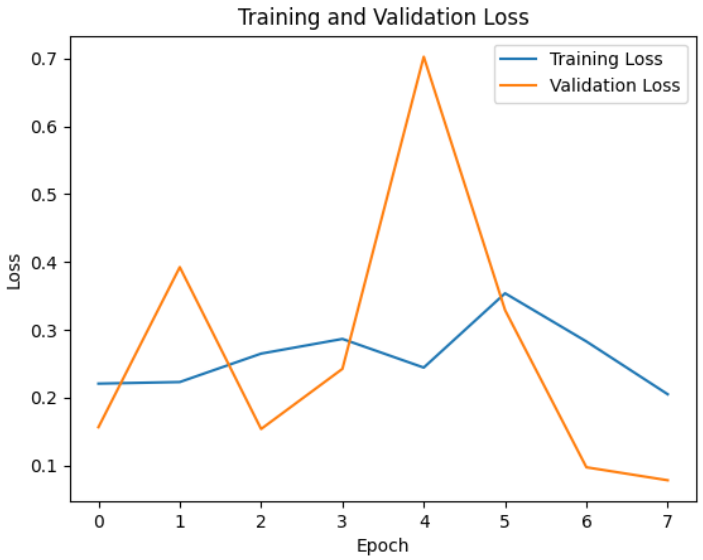

Figure 17: Training and validation loss curve (after adding type 2 and 3 images)

Based on the assumption that training data is still insufficient, we then increase total training data by 140% by adding type 4 and 5 wound images, type 4 replaced the background with real skin texture, making it more like real wounds, and type 5 making wound's border blurry and as we hope the model to learn how to extract the overall shape of the wound without interfering by the unclear border. The training and validation loss shows that it was a successful strategy as shown in Fig.15. The training and validation loss both decrease between 0 to 9 epochs, and then validation loss increases significantly, meaning overfitting of the training.

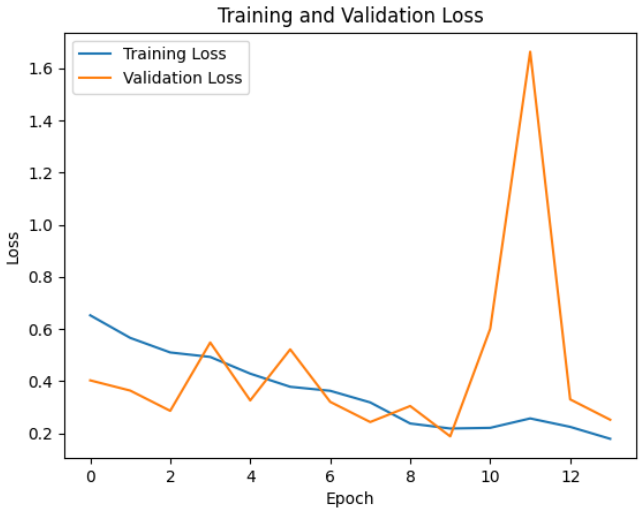

Figure 18: Training and validation loss curve (after adding type 4 and 5 images)

2.2.9.3 Making corresponding masks for wound images

Mask is an important part of the training data, each image needs to have a corresponding mask. A mask refers to a binary image consisting of 0 and 1, it can be used to indicate and highlight regions of interest in the image. Making a mask for images could be extremely time consuming as it requires manually outline the border of the wound using pen-like tools as shown in Fig.16. The annotation tool that was used is Labelbox, which provides free annotation tools for students for the first 5,000 images. The Labelbox uses .JSON format to store the mask's information, which means we are unable to see a mask as an image directly, it is only visible to us when running the code that transfers the JSON content into an image. Fig.17 shows a function that transforms JSON files into masks images. The requests.get(polygon) retrieves the mask image from the URL, and Image.open(BytesIO(response.content)).convert('L') opens and converts the image into grayscale (the 'L' mode in Pillow stands for grayscale). The grayscale image is then converted into a numpy array to form the mask.

Throughout the project, we made over 500 masks, not all the masks will eventually appear to be the same shape as the drawing shape, some types of images will easily fail and require repeated annotation works.

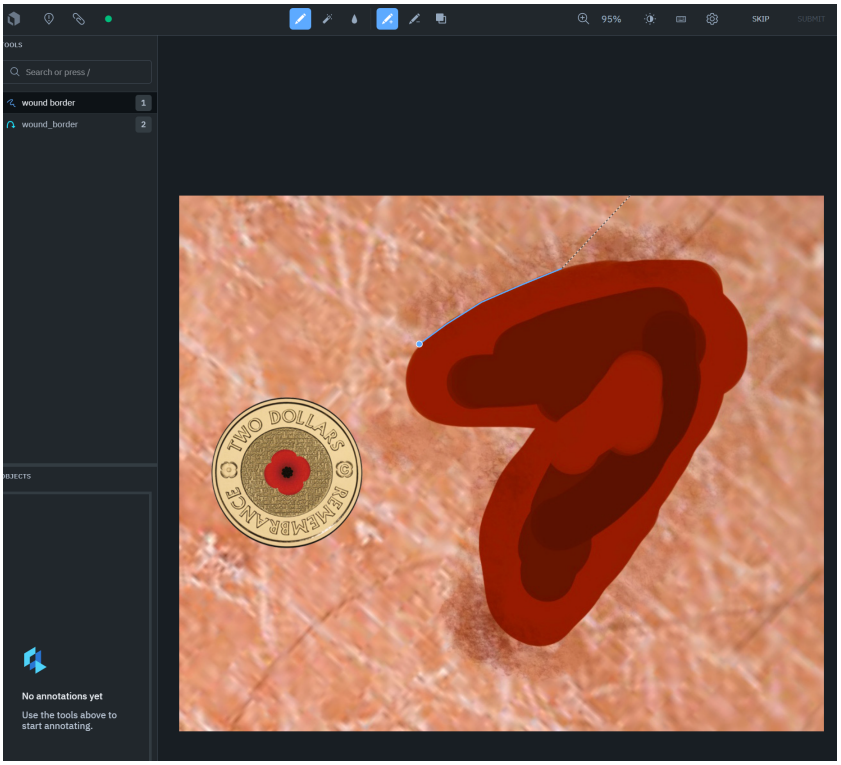

Figure 19: Making a mask for an image

```python
def load_image_and_mask(file, images_json_path, masks_json_path):
    with open(os.path.join(masks_json_path, file[:-5] + '.json')) as f:
        mask_json = json.load(f)

    if 'Label' in mask_json:
        image = cv2.imread(os.path.join(images_json_path, file[:-5] + '.jpg'), cv2.IMREAD_COLOR)
        if image is None:
            print(f"Unable to read image file {file}. Skipping...")
            return None, None
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

        polygon = mask_json['Label']['objects'][0]['instanceURI']
        response = requests.get(polygon)
        mask = np.array(Image.open(BytesIO(response.content)).convert('L'))

        return image, mask
    return None, None
```

Figure 20: Loading mask from JSON files

2.3 Solution 2 Extract outline as wound borders

2.3.1. Assumptions

- All images are taken with a 2 dollar coin as a reference object, and the object is placed in a pre-selected location which is at the same level as the wound and near the wound.
- All images are taken at a good angle.
- All images are taken in good light.
- The doctor will draw the wound outlines as a pre-process of the wound images using blue colour.

2.3.2 Extract outline

To extract outlines of wounds in the image, an extract_blue_contour function has been defined, which extracts the outline of the blue-coloured regions in the image, essentially useful in identifying wound areas in medical images.

The function begins by loading an image using the file path provided. It then gets the dimensions of the image and checks if the height or width is larger.

The next step involves resizing the image to a standard size of 256 pixels on the longest side while maintaining the original aspect ratio. To make sure the image is still square, any additional space is padded. This standardisation is crucial for image processing, which often requires input data to be in a consistent format.

The function then converts the image from the BGR (Blue, Green, Red) colour space to the HSV (Hue, Saturation, Value) colour space. This conversion is done because the HSV colour space often simplifies colour-based segmentation tasks.

Once the image is in the HSV colour space, the function defines a colour range for blue. It then creates a 'mask' that isolates pixels within this blue colour range in the image.

The function then identifies contours, which are essentially boundaries of connected pixels with the same colour, in the blue-masked image. It draws these outlines on a new image, contour_image, that's initially all white.

After that, it creates a binary mask, wound_area, where the contour area is filled in white and the rest of the image is left black. This binary mask can provide a simplified view of the wound area, with the wound area in white and non-wound areas in black.

The function then counts the number of white pixels, which represent the wound area, and calculates its ratio to the total number of pixels in the resized image, which is provided for further scaling to get real wound size.

At the end of the function, it returns the contour image, the pixel ratio (which signifies the wound's relative size), and the binary mask of the wound area.

```python
def extract_blue_contour(image_path):
    image = cv2.imread(image_path)
    if image is None:
        raise ValueError(f"Unable to read image file {image_path}. Check file path/integrity")

    # Get the dimensions of the original image
    original_height, original_width = image.shape[:2]

    # Determine whether the original image's height or width is bigger
    if original_height > original_width:
        long_side = original_height
        short_side = original_width
    else:
        long_side = original_width
        short_side = original_height

    # Resize the image while maintaining its aspect ratio
    target_size = 256
    image = resize_with_aspect_ratio(image, target_size)
    image = pad_image(image, target_size)

    hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Define the lower and upper boundaries for the blue color
    lower_blue = np.array([100, 50, 50])
    upper_blue = np.array([130, 255, 255])

    # Create a mask that isolates the blue color in the image
    blue_mask = cv2.inRange(hsv_image, lower_blue, upper_blue)

    # Perform contour detection
    contours, _ = cv2.findContours(blue_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Draw the largest contour on a white background
    contour_image = np.zeros_like(image)
    contour_image.fill(255)
    cv2.drawContours(contour_image, contours, -1, (0, 0, 0), 2)

    # Create a binary mask where the contour with wound region is white (255) and the rest of the image is black (0)
    wound_area = np.zeros(image.shape[:2], dtype=np.uint8)
    cv2.drawContours(wound_area, contours, -1, (255, 255, 255), thickness=cv2.FILLED)

    # Count the pixels inside the filled contour
    pixel_count = np.count_nonzero(wound_area)

    # Get the ratio of the resized image to origical image
    resize_ratio = 256 / long_side
    resize_short_size = resize_ratio * short_side

    # Calculate the total number of pixels in the resized image
    total_pixels = 256 * resize_short_size

    # Calculate the ratio of pixels inside the filled contour to the total pixels in the resized image
    pixel_ratio = pixel_count / total_pixels

    return contour_image, pixel_ratio, wound_area
```

Figure 21: Extract blue outline

2.3.3 Resize masks back to original

In a machine learning and image processing context, it's often necessary to resize images and their associated masks to standardise their dimensions. This allows the models to process them more efficiently. However, once the processing is done, it's important to resize these masks back to their original size for a few reasons, which will be explained in a simplified manner.

Primarily, resizing the mask back to the original image size allows for a more accurate overlay of the original image. This is particularly important in our tasks, where precise localization of wound regions is required. A mask that is not accurately resized could misrepresent the wound's location or its extent, leading to inaccurate measurements and assessments.

Moreover, maintaining the original size is essential for visual inspection and interpretation. Healthcare professionals who are evaluating the wounds are familiar with the original image's context, and a resized mask might look out of place or be misleading. Hence, ensuring the mask aligns perfectly with the original image makes it more interpretable and reliable for further analyses.

Finally, one of the most significant reasons for this is to ensure that the size of the wound can be measured correctly. If the mask remains in the resized state, any wound size measurements derived from it would not reflect the real dimensions on the original image. This could result in significant inaccuracies when assessing the wound's size, which could potentially impact clinical decision-making.

Resizing the mask back to the original image size aligns it perfectly with the original image's scale. As a result, the measurements of wound size derived from this mask will accurately represent the wound's dimensions in real-life. This ensures that clinicians have precise and reliable information when evaluating the wound's state and progress, leading to better patient care.

This part contains two function definitions: remove_padding and resize_to_original. Both functions are used in image processing to manipulate the sizes of images and masks.

The remove_padding function is designed to take an input image and its original dimensions and remove any padding that has been previously added to standardise the image's size. Padding refers to the process of adding extra pixels around the image to make it a certain size.

First, the function calculates the resize ratio, which is a factor by which the original image has been resized. It uses this ratio and the original dimensions to calculate how much padding was added to the image. It distinguishes between two cases: if the original image's height was larger than its width and vice versa.

The function then uses this padding information to remove the extra pixels, thus restoring the image to its original size. If the image is three-dimensional (which means it has colour channels), it applies this process to all the channels. If the image is two-dimensional (grayscale), it only needs to apply this process once.

The resize_to_original function simply takes the mask image without paddings and resizes it back to its original dimensions. It does this by using the OpenCV function cv2.resize, which can resize an image to specified dimensions.

In summary, these two functions are a part of the process of reverting the images back to their original format after they've been resized and padded for the purposes of image processing and machine learning model input. They ensure the output masks can be overlaid correctly on the original images and are more convenient for further measurement.

```python
# Remove padding to output mask to store original image size mask
def remove_padding(image, original_width, original_height):
    height, width = image.shape[:2]

    if original_height > original_width:
        resize_ratio = height / original_height
        resize_width = int(resize_ratio * original_width)
        left_pad = (width - resize_width) // 2
        right_pad = width - resize_width - left_pad
        top_pad =0
        bottom_pad = 0
    else:
        resize_ratio = width / original_width
        resize_height = int(resize_ratio * original_height)
        top_pad = (height - resize_height) // 2
        bottom_pad = height - resize_height - top_pad
        left_pad = 0
        right_pad = 0

    if image.ndim == 3:
        return image[top_pad:height - bottom_pad, left_pad:width - right_pad, :]
    elif image.ndim == 2:
        return image[top_pad:height - bottom_pad, left_pad:width - right_pad]
    else:
        raise ValueError("The input image must have 2 or 3 dimensions.")

# Resize the binary mask to the original image size
def resize_to_original(image, original_width, original_height):
    return cv2.resize(image, (original_width, original_height))
```

Figure 22: Resize the mask back to original size

2.3.3 Evaluate the solution

In order to evaluate the solution, a contour folder which includes six artificial wound images and fourteen real wound images from a client with blue outlines has been created.

Then a python function named display_binary_mask_from_outline is designed to visually display original images and the corresponding binary masks that represent wound areas within these images.

Firstly, it loops through every file in a specified directory (input_directory). This directory is the contour folder.

Secondly, for each file in the directory, it checks whether the file is a JPEG image file. It does this by checking if the file name ends with the '.jpg' extension. If it does, the function continues processing that file; if not, it moves to the next file. If the file is a JPEG image, the function constructs the full file path by combining the directory path with the image file name.
Thirdly, using this file path, the function then calls another function extract_blue_contour(input_path), which takes in an image file path, processes the image and gets the binary mask of it.

Fourthly, after getting the numpy array of binary masks, the function then displays the original image and the wound area (which is a binary mask representing the wound in the image). It uses the OpenCV “imshow” function for this, and titles each window with the respective image file name.

Finally, once all images have been processed and displayed, the function destroys all the windows created using the OpenCV destroyAllWindows function.

The following figures display the functions and the results of binary masks with the original wound images. And it is clear that the solution is effective and accurate.

```
def display_binary_mask_from_outline(input_directory):
    for image_file in os.listdir(input_directory):
        if image_file.endswith('.jpg'):
            input_path = os.path.join(input_directory, image_file)
            contour_image, pixel_count, pixel_ratio, wound_area, original_image = extract_blue_contour(input_path)

            # Display the output image
            cv2.imshow(f'Original Image {image_file}', original_image)
            cv2.imshow(f'Wound Image for {image_file}', wound_area)
            cv2.waitKey(0)

    cv2.destroyAllWindows()
```

Figure 23: Display the results

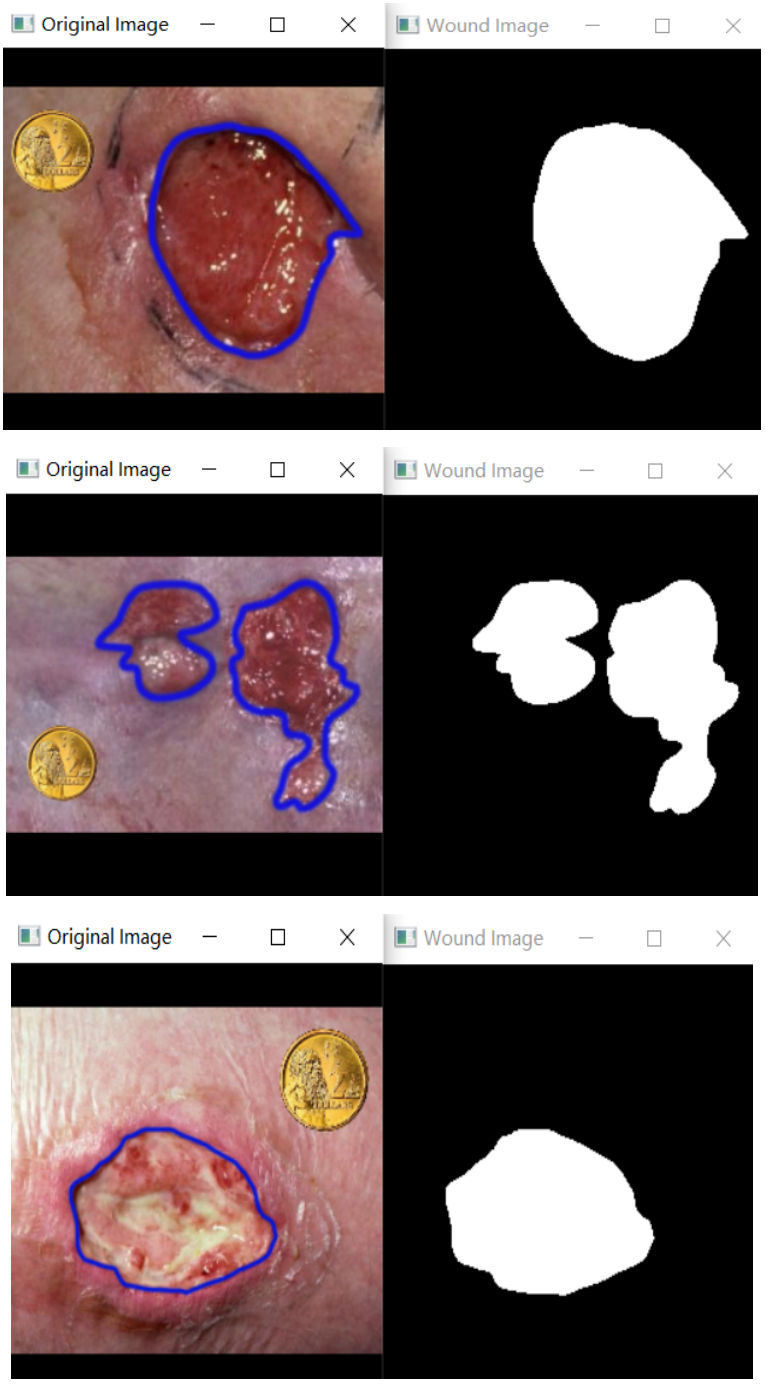

Figure 24: Display the masks with the wound images

3. Coin detection and Wound measurement

3.1 Purpose and method

The purpose of coin detection is to use the coin as a reference to hence calculate the actual wound size. The input is the image with a two-dollar coin and wound area. Also we use some data outputs from the wound detection part, for the optimization of the algorithm.

The project development environment is Python. OpenCV and some modern image processing algorithms are used.

3.2 Pre-process image

Before performing item detection in OpenCV, the typical workflow involves a series of image pre-processing steps. The following techniques are used in our project.

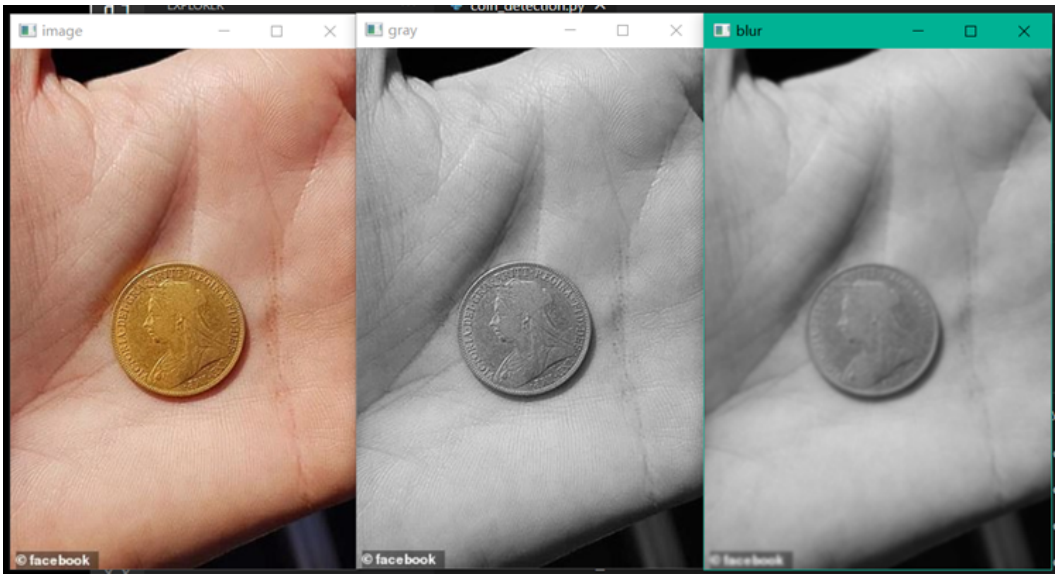

Figure 25 Grayscale and blurring for image pre-processing.

1. Grayscale conversion:

Grayscale images are good to use in computer level tasks as it reduces the computational complexity while not sacrificing the information in the image. The grey method simplifies the image by reducing the image to a single channel and representing the intensity of each pixel

.

2. Blur:

Blurring, also known as smoothing. The noise in an image can impact on the following item detection steps and lead to false detection. The blurring method helps to reduce noise and hence remove small details in the image.

3. Canny edge detection:

After grayscale and blurring, the image's edge and boundaries need to be identified. The edge detection is crucial for coin detection because the coin can be easily defined by the coin boundary. The Canny edge detection algorithm helps identify the boundary in an image and provide a binary image where edges are represented in white pixels, and the background in black.

In the canny detection algorithm, the params could change the quality of the result. For example, in the image above, I used Canny edge detection with the threshold values 90 and 255. If the edges in the image are not strong enough, the lower threshold value might be too high, causing the Canny function to miss some edges. In the following steps, I will try to adjust the threshold values to retain more edge information.

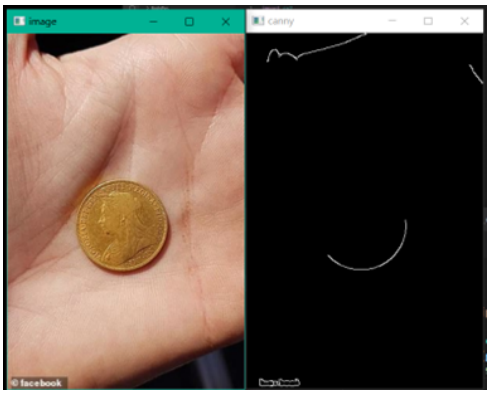

Figure 26 Result for Canny edge detection params setting 1

```
canny = cv2.Canny(blur, 90, 255)
```

Figure 27 Canny edge detection params setting 1

Above figure 26/27 shows the params and the result. 90 is the lower threshold value, the algorithm links edges that have a gradient intensity higher than this threshold. 255 is the higher threshold value; the edges that have a gradient intensity higher than this value are treated as a strong edge. As we can see from the figure, the circle edge detected is not complete, hence I reduced the lower threshold to 10 and saw the result.

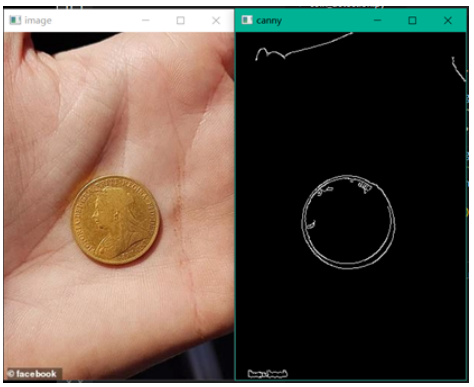

Figure 28 Result for Canny edge detection params setting 2

```
canny = cv2.Canny(blur, 10, 255)
```

Figure 29 Canny edge detection params setting 2

Above figure 28/29 shows the params and the result. 10 is the lower threshold value. 255 is the higher threshold value. It is obvious that reducing the ratio between threshold 1 and threshold 2 will result in more detected edges.

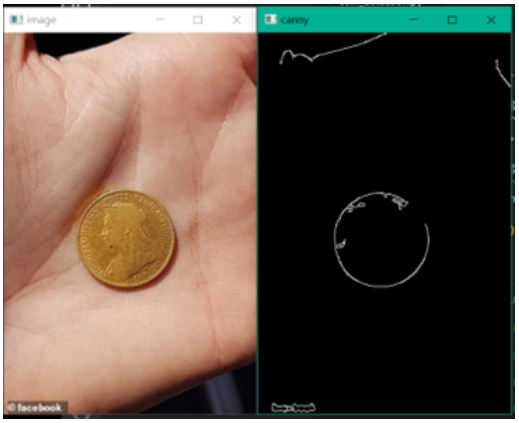

Figure 30 Result for Canny edge detection params setting 3

```
canny = cv2.Canny(blur, 12, 255)
```

Figure 31 Canny edge detection params setting 3

Figure 30 shows the best result I can get after adjusting the params. It is proven that adjusting the Canny param can effectively improve the result of detection. Hence, in the code, the params of Canny are scaling in a loop as shown in figure 32/33. In that way, more combinations of params can be tested to find the best circle.

```
while best_circle is None and attempt < max_attempts:
    circles = find_circles(image, mask, param2_start=50, max_attempts=40, cannyParams= 30-attempt*2)
    attempt += 1
```

Figure 32 code to find the best params for Canny

```
canny = cv2.Canny(blur, cannyParams, cannyParams*3)
```

Figure 33 Canny params that iterates 'attempt' times

However, by only adjusting the params of the Canny algorithm, the best result is not even 100 percent ideal for our detection purpose. I also can adjust the Gaussian blur parameters to get a better result. The main purpose of applying a Gaussian blur before edge detection is to reduce noise in the image. Noise can produce false edges and may affect the accuracy of the detected edges. By tuning the blur parameters, I can achieve a better balance between noise reduction and edge preservation.

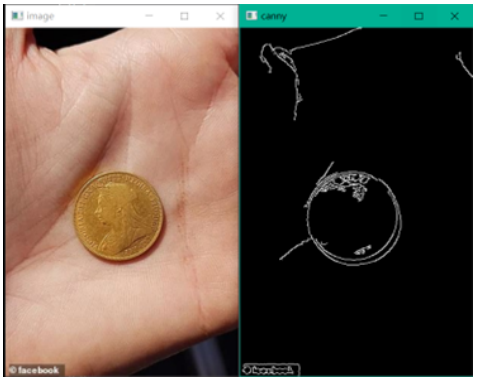

Figure 34 Result for Gaussian blur params setting 1

```
blur = cv2.GaussianBlur(gray, (9,9),1)
```

Figure 35 Gaussian blur params setting 1

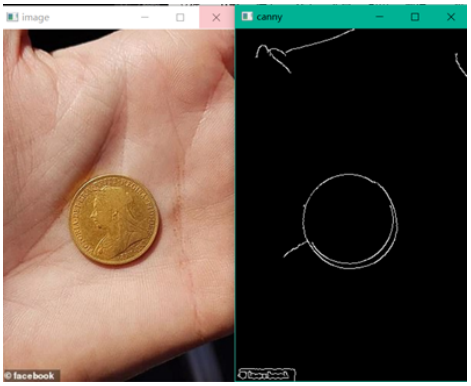

Figure 36 Result for Gaussian blur params setting 2

```
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(gray, (7,7),1.2)
canny = cv2.Canny(blur, 40, 255)
```

Figure 37 Gaussian blur params setting 2

Above figure 34 - 37 show the result before and after adjusting the Gaussian blur parameters. The params (7,7) and (9,9) refer to the kernel size of the Gaussian filter.

The number is the size of the kernel cell used for blurring. The higher the number, the stronger the blur. In the above example, I reduced the kernel size, hence more details of the image should be detected. The last number is the standard deviation of the Gaussian kernel. A higher number will result in more blurring. In the above example, the higher standard deviation leads to a result with critical lines and without unnecessary details.

```
# Check if any circles were found after filtering
# if circles is None or len(circles) == 0:
    # Change the Canny parameters
blurParams = blurParams - 0.2
```

Figure 38 The params of blurring function changes in a loop

```
while attempt < max_attempts:
    blur = cv2.GaussianBlur(gray, (9, 9), blurParams)
```

Figure 39 Blurring function that contains the changing params

Hence, in the code, the parameters of the blurring function are embedded in a loop to find the best combination of params. As shown in figure 38, if no circle satisfies the condition of being the right coin, the params of the blurring function will be changed and the function will be called again to find the circle.

3.3 Detect all possible coins

In this step, the algorithm is guided to find all possible coins. The computer does not know what a coin is, it only recognizes shapes and edges. Thus, the core concept of this step is to help the program find the shapes or edges that best match the description of a coin. Based on one of the assumptions of our project, the coin is assumed as a two-dollar coin, and may be updated in the future.

The HoughCircles function in OpenCV is commonly used for coin detection due to its ability to detect the circular shapes in an image. It has some advantages as follows:

1. Robustness to noise: The coin does have distinct shapes. However, when a patient captures the image, the image may contain noise, uneven lighting or some other interference. The HoughCircle method is robust to such noise and can accurately detect circles in images with disturbances.

2. Detects circles accurately: The HoughCircle method uses the Hough Transform technique, which is specifically designed to detect circular shapes. It can also handle incomplete circles, different sizes of circles, and even discontinued edge circles.

3. Efficiency: The underlying algorithm of HoughCircle is somehow complex, the HoughCircle itself is generally efficient and is able to provide real-time performance depending on the image size and hardware capabilities. In our project, the patient may want to see the result right away. Thus, efficiency is an important benefit to our project.

4. Parameter flexibility: The HoughCircle method provides several parameters for us to control the detection process.

When all the circles are found, these circles are treated as all possible coins. In the following coding stage, I used an OpenCV tool to contour the coin shape out to help visualise the result.

```
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
blur = cv2.GaussianBlur(gray, (7,7),1.2)
canny = cv2.Canny(blur, 50, 255)

# Detect circles using HoughCircles
circles = cv2.HoughCircles(canny, cv2.HOUGH_GRADIENT, 1, 20, param1=50, param2=30, mi

if circles is not None:
    circles = np.uint16(np.around(circles))
    for i in circles[0, :]:
        # Draw the outer circle
        cv2.circle(img, (i[0], i[1]), i[2], (0, 255, 0), 2)
```

Figure 40 Code snippet to help contour the shape of circles out

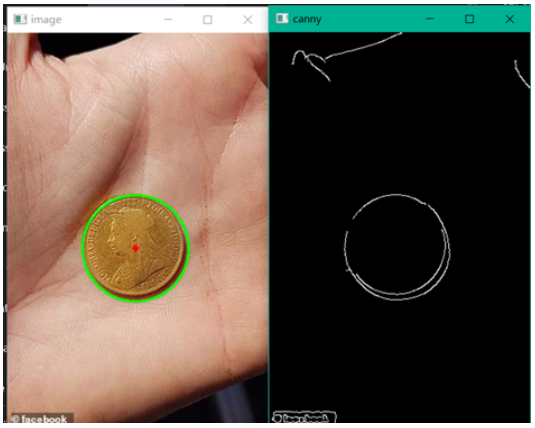

Figure 41 Coin detected is visualised with contour

As shown in figure 41, the contour is added to the boundary of the coin detected. In the following development steps, this contour helps me to identify the correctness of finding the coin and hence modify the code.

3.4 Find the correct coin

Based on the assumption, there will only be one coin in the image. In case of detecting two circles, I need the program to select the best circle as a result. After finding all the possible circles, the next step is to determine which circle is the right coin. The algorithm has been tested and modified in some cases progressively.

3.4.1 Improvement in the first case

The first case is the image with the coin only. In this scenario, the code is tested to find and determine only one circle as the correct coin. The params are adjusted to get the best result.

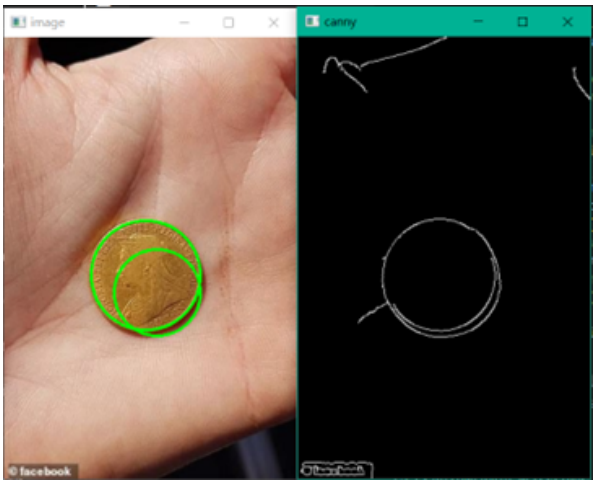

Figure 42 It is possible to find several circles

In this scenario, adjusting the params in HoughCircle can effectively find the best result.

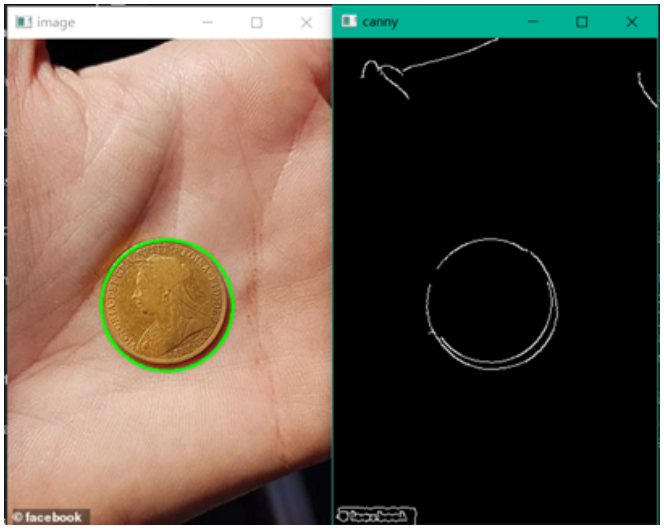

Figure 43 Best result in scenario 1 after adjusting the params in HoughCircle

3.4.2 Improvement in the second case

The second case is the image with the wound and coin. After the teammates update their source of images for model training, the coin detection function is tested again. Some issues found in detection. In the below example figure 44, there are five circles detected (shown in the image on the left), the algorithm chose the wrong one as the best circle (shown in the image on the right).

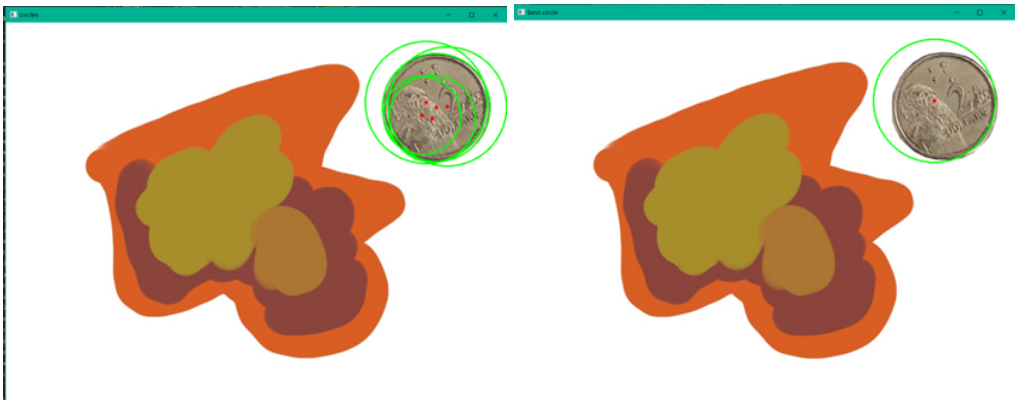

Figure 44 Origin result in scenario 2, the wrong circle is chosen

The idea to solve this issue is to add a function to help the system compare the average colour from inside and outside of the circle. By comparing the difference, whichever has the biggest difference colour average will be chosen as the best circle.

```
def get_average_color(image, x, y, radius, inside=True):
    mask = np.zeros(image.shape[:2], dtype=np.uint8)
    cv2.circle(mask, (x, y), radius, 255, -1 if inside else 1)
    mean_val = cv2.mean(image, mask=mask)
    return mean_val
```

Figure 45 Code snippet to calculate the average colour inside a circle

```
inner_average_color = get_average_color(image, x, y, radius, inside=True)
outer_average_color = get_average_color(image, x, y, radius, inside=False)

difference = sum(abs(inner_average_color[j] - outer_average_color[j]) for j in range(3))

if difference > best_difference:
    best_circle = i
    print(f"difference now: {difference}")
    best_difference = difference
    print(f"best difference now: {best_difference}")
```

Figure 46 Code snippet to select the best circle based on the average colour

As shown in figure 45, the average colour of the area inside and outside the circle is calculated, the colour difference is stored. As shown in figure 46, the algorithm will choose the circle that has the largest colour difference as the coin.

Figure 47 Result after modifying code in the second case

Now we check the result again, it is shown in figure 47 that the code is able to find the coin correctly.

3.4.3 Deal with background noise

At the beginning, the params of the HoughCircle function are set to a certain set of numbers. However, while testing the functionality against the image with background noise added in, the function was found to have around a 30% chance of failure.

```
PS E:\Github\Woud_Detection_Machine_Learning> & E:/Anaconda/python.exe e:/Github/Woud_Detection_Machine_Learning/v1_coin.py
cannot find circles.
```

Figure 48 Failure case: cannot find any circle.

After investigating the issue, I found that it might be due to the params in the circle detection function being fixed. Now I will add a loop to let the system dynamically change the params if no circles are found.

```
best_circle = None
attempt = 0
max_attempts = 10
while best_circle is None and attempt < max_attempts:
    circles = find_circles(image, mask, param2_start=50, max_attempts=40, cannyParams= 30-attempt*2)
    attempt += 1
```

Figure 49 Code snippet to loop different params of circle finding function

As shown in figure 49, when the best circle is not found in the function, the params of the function will change and the circle finding function will be called again until it finds the best circle we are looking for.

There can be another edge case, when the algorithm finds only one circle, the algorithm will select this circle as the best circle (coin). However, sometimes the circle found is not the right coin. Hence, the algorithm needs to find more than one circle before determining which one is the best.

```python
def find_circles(image, mask, param2_start, max_attempts, cannyParams):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    circles = None
    attempt = 0
    blurParams = 1.5

    while circles is None or len(circles) <= 2:
        while attempt < max_attempts:
            blur = cv2.GaussianBlur(gray, (9, 9), blurParams)
            canny = cv2.Canny(blur, cannyParams, cannyParams*3)
            # # Apply the mask to the Canny output
            masked_canny = cv2.bitwise_and(canny, cv2.bitwise_not(mask))
```

Figure 50 Code snippet to find more than two circles before selecting the best circle

As shown in figure 50, the params will keep looping if no more than two circles are found. This makes sure the algorithm will have enough samples to compare and select from before determining the coin circle.

Now it can correctly find the best circle after dynamically searching the circles with different params, shown in figure 51.

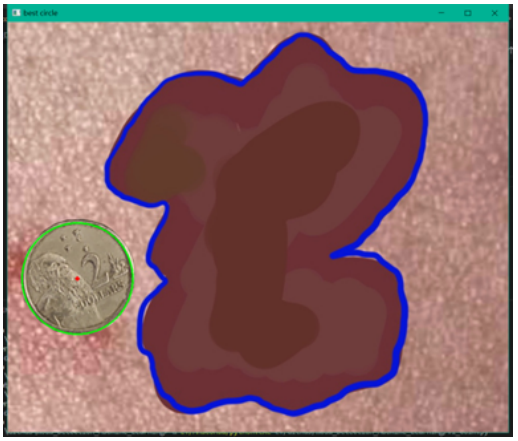

Figure 51 Result after looping the find_circles function

However, there can be another edge case that the best circle found is inside the wound area, as shown in figure 52 below. This issue is caused by the function that has used the full image to detect the circles.

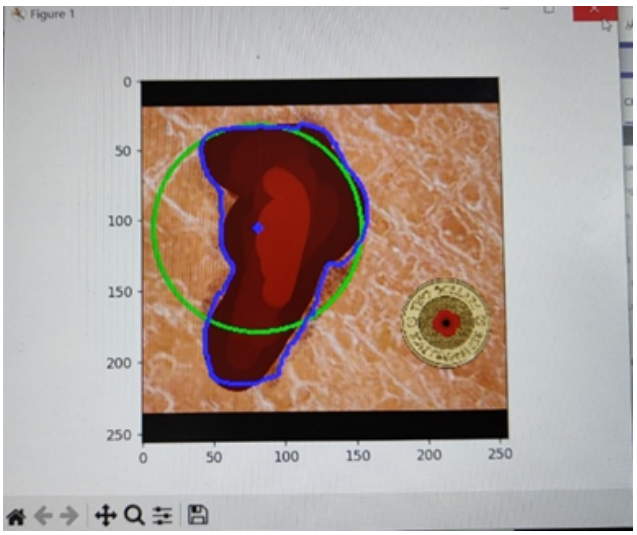

Figure 52 Edge case: Circle detected inside the wound

To address this issue, I imported one of the output data from the wound detection part that was done by my teammate. The data I imported is the wound area binary mask.

```
# # Apply the mask to the Canny output
masked_canny = cv2.bitwise_and(canny, cv2.bitwise_not(mask))

cv2.imshow('masked canny', masked_canny)
cv2.waitKey(0)
```

Figure 52 Code snippet to apply wound mask on the image

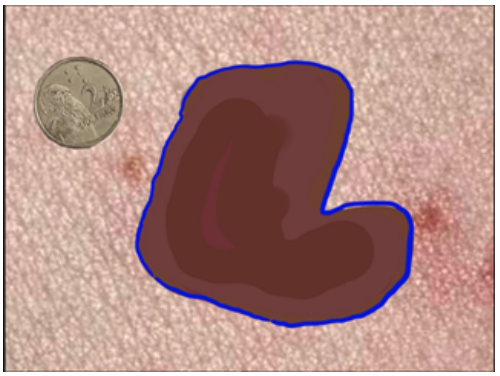

Figure 53 Image before applying the wound mask

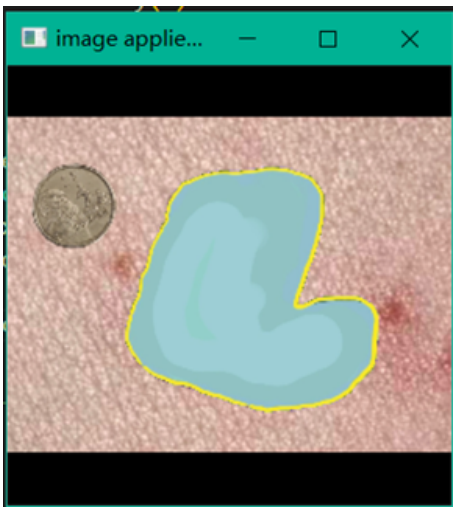

Figure 54 Image after applying the wound mask

As shown in figure 54, the image applies the wound binary mask and replaces the original colour of wound to avoid any interference that may lead to a bad result.

Then, I also need to recognize the circles that intersect with the wound area. The idea of finding these circles is to calculate the distance from the centre of the circle to the wound edge and compare it with the radius of the circle.

```
# Filter out circles that intersect with the mask
filtered_circles = []
if circles is not None:
    for circle in circles[0, :]:
        x, y, radius = circle

        # Ensure the circle is inside the image boundaries
        if x - radius < 0 or y - radius < 0 or x + radius >= image.shape[1] or y + radius >= image.shape[0]:
            continue  # Circle is outside image, skip it

        circle_mask = np.zeros_like(mask)
        cv2.circle(circle_mask, (int(x), int(y)), int(radius), (255, 255, 255), -1)

        # Check if the circle is inside the mask
        intersection = cv2.bitwise_and(circle_mask, mask)
        if np.sum(intersection) > 0:
            continue  # Circle intersects with mask, skip it
        # Circle does not intersect with mask, add it to filtered circles
        filtered_circles.append((x, y, radius))
        print(f"filtered_circles each:{filtered_circles}")

    # Convert filtered_circles to a NumPy array
    filtered_circles_array = np.array(filtered_circles)

    # Reshape the array to have the same format as circles
    circles = filtered_circles_array.reshape((-1, 1, 3))
    print(f"filtered_circles final :{circles}")

    if len(circles) >= 2:
        return circles
```

Figure 55 Filter out circles that intersect with the wound area

In figure 55, the code snippet finds all the circles that do not intersect with the wound area and puts them into a result array. I also tested the result on other images, below is an example of the result in figure 56/57.

Figure 56 All possible circles detected

Figure 57 Best circle found after applying filter

In figure 56, we can see the algorithm detected three possible circles as coins. In figure 57 we can see that after applying the filter, the correct circle is selected as the coin result.

The code was also tested against all the image sources we have, below are some outputs I got.

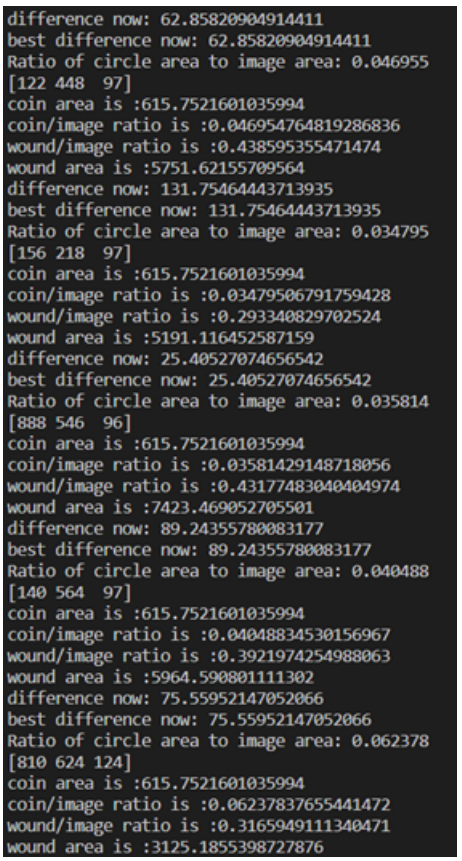

Figure 58 The code is now able to find the correct coin in all testing images

Based on the output in figure 58, the code is revealed to be successfully finding the coin for every image.

3.5 Output for actual wound area and calculation

Some data are returned as output for further development. Also, some output can be used for wound area calculation.

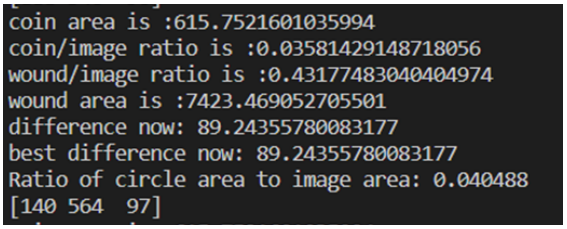

Figure 59 Part of the output from coin detection function

The idea of wound area calculation is to get the ratio of coin to image, and the ratio of wound to image. As we have the actual size of the coin, it is easy to calculate the actual size of the wound by using the data we have in figure 58.

The important data that the patient is looking for can also be displayed on the image.

Figure 60 Final result image with actual wound size

As the calculation of wound to image ratio uses the output from wound detection function, another teammate finds the wound mask output from the wound detection function might not be correct. Hence, some further improvement on the wound area calculation is applied and will be explained in detail from the next section.

4 Improving Measurement Accuracy

4.1 Reviewing Existing Measurement Functions

As mentioned at the end of the previous section, the initial measurement functions that were developed were not as accurate as we had hoped. This was initially identified as the project providing an incorrect real-world measurement for the reference object, however even after correcting this, the existing functions still returned unexpected results.

This section covers the development of new and improved measurement functions that provide more accurate results, as well as a new set of visualisations to assist in interpreting the results efficiently for the user.

4.2 Measuring Objects Extracted from Images

After the wounds and the reference object have been extracted as binary masks using the techniques detailed in the previous sections, they can then easily be converted to OpenCV data structures to calculate their measurements.

To improve the measurement process, I also refactored the function which extracts outlined wounds from images, which both simplifies the process, as well as ensures

that only the most valuable data is returned, which can be seen in Figure 61.

```
# Extracts a contour from an image which has a blue outline drawn on it
def extract_contours_from_outlined_image(image):
    hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

    # Define the lower and upper boundaries for the blue color
    lower_blue = np.array([100, 50, 50])
    upper_blue = np.array([130, 255, 255])

    # Create a mask that isolates the blue color in the image
    blue_mask = cv2.inRange(hsv_image, lower_blue, upper_blue)

    # Perform contour detection
    contours, _ = cv2.findContours(blue_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    # Return list of contours
    return contours
```

Figure 61 - Refactored Function for Extracting Outlined Wounds

As this data was extracted from images, the only unit of measurement which can be calculated directly is pixels, so the first step was to develop a series of measurement functions which calculated the dimensions of circles and contours in pixel-space. The valuable measurements for a circle were identified as the area and the radius, and the area and the X and Y lengths for contours.

As the pixel radius of the circle was already calculated during the extraction process, it can be used to calculate the area of a circle in pixels². This calculation is as simple as multiplying the squared radius by Pi, the code for which can be seen in Figure 62.

```
# Get the area of a circle in pixels^2
def get_circle_area_px(radius_px):
    return math.pi * (radius_px ** 2)
```

Figure 62 – Calculating the Area of a Circle in Pixels²

Calculating the area of a wound contour was an area that proved to be difficult in the early stages of the project, with the initial solution not providing accurate results. However, while researching ways to improve the accuracy, it was discovered that OpenCV already contains a function to return the area in pixels², which greatly simplified this step as seen in Figure 63.

```
# Get the area of a contour in pixels^2
def get_contour_area_px(contour):
    area_px = cv2.contourArea(contour)

    return area_px
```

Figure 63 - Calculating the Area of a Contour in Pixels²

Measuring wound lengths was not something we had initially intended to do. However, it was decided that while area measurements are useful, lengths and widths are more commonly used measurements and may be more useful in practical application. The process used to calculate the lengths was to compute a bounding box around the contour. This was achieved using OpenCV's *MinAreaRect* function, which computes a bounding box which is rotated to an angle where the box has the smallest possible dimensions around the contour area. From there it is as simple as measuring the distance between points on the X and Y axis relative to the box. Non-relative angles could be used if the *BoundingRect* function was used instead. This function also computes a box around a contour but does not use rotation to find the smallest possible box. The code for these measurements can be seen in Figure 64.

```
# Get the X and Y dimensions of a countour in pixels
def get_contour_size_px(contour):
    # Calculate a rect around the contour.
    rect = cv2.minAreaRect(contour)

    # Covert the rect to a list of points
    box = cv2.boxPoints(rect)
    box = np.int0(box)

    # Get the four corner points of the box
    top_left, top_right, bottom_right, bottom_left = box

    # Calculate the pixel legnths of each axis
    x_length_px = get_length_px(top_left, top_right)
    y_length_px = get_length_px(top_left, bottom_left)

    return x_length_px, y_length_px
```

Figure 64 - Calculating the X and Y Lengths of a Contour in Pixels[2]

4.3 Converting Pixel Measurements to Real-World Units

Once the pixel measurements of the objects have been calculated, they can then use the radius of the reference object to convert these measurements into a real-world metric, which for this project is millimetres. The fundamental principle to this conversion is what we have been calling the *pixels-per-millimetre* ratio. The equation for calculating this ratio as defined by Rosebrock (2021) is to divide the pixel size of the reference object by its known size in millimetres, as seen in Figure 65.

The resulting ratio can then be used to convert any pixel measurements to millimetres by dividing the pixel measurement by the ratio, as seen in Figure 66.

$$pixels\_per\_metric = \frac{object\_width}{known\_width}$$

Figure 65 - Pixels-per-Metric Equation (Rosebrock 2021)

$$real\_world\_width = \frac{object\_width}{pixels\_per\_metric}$$

Figure 66 - Converting Pixel Measurements to Millimetres (Rosebrock 2021)

The functions to calculate the pixels-per-metric ratio, and its related conversions were similarly simple. However, the concept seemed difficult for some members of the team, so additional functions were also developed in an attempt to make their usage clearer.

The main function for these tasks is one which converts pixels to millimetres (Figure 66) using the equation defined in Figure 65 above, however an exponent parameter was added as squaring the ratio is required for calculating area measurements. Additionally, as the calculation of the ratio is similar to the conversion equations, a helper function was created to distinguish the two tasks (Figure 67), however it actually just routes to the *pixels_to_millimetres* function.

```
# Convert pixel measurements to millimetres
# NOTE: Set exponent=2 if converting area measurements!
def pixels_to_millimetres(pixels, pixels_per_millimetre_ratio, exponent=1):
    return pixels / (pixels_per_millimetre_ratio ** exponent)
```

Figure 67 - A Function to Convert Pixels to Millimetres

```
# Calculate the pixels-per-millimetre ratio
# This function is just to avoid confusion as pixels_to_millimetres() could be used directly
def calculate_pixels_per_millimetre_ratio(circle_radius_px, reference_radius_mm):
    return pixels_to_millimetres(circle_radius_px, reference_radius_mm)
```

Figure 68 - The Function to Calculate the Pixels-per-Millimetre Ratio

For the sake of creating a complete suite of measurement and conversion tools, a function was also created to convert measurements in millimetres back to pixel measurements (Figure 68) as well as a function to convert the real-world reference object measurements to pixels (Figure 69). These two functions were not used in the main project but proved useful for unit testing to validate the accuracy of the measurement and conversion functions.

```
# Convert millimetre measurements to pixels
# NOTE: Set exponent=2 if converting area measurements!
def millimetres_to_pixels(millimetres, pixels_per_millimetre_ratio, exponent=1):
    return millimetres * (pixels_per_millimetre_ratio ** exponent)
```

Figure 69 - A Function to Convert Millimetres Back to Pixels

```
    # Get the radius of a circle in pixels^2
    def get_circle_radius_px(radius_mm, pixels_per_millimetre_ratio):
        return millimetres_to_pixels(radius_mm, pixels_per_millimetre_ratio)
```

Figure 70 - A Function to Convert Coin Measurements to Pixels

4.4 Final Measurement Functions

The measurement functions used to calculate the final results are, in essence, helper functions that combine the functions listed above to generate accurate measurements for wounds and reference objects in millimetres. As noted above, there are two types of data which require measurement: Contours and circles, both of which come from the OpenCV library. Due to this, there are again two sets of measurement functions, a set for measuring contours (Figure 70 & Figure 71), and a set for measuring circles (Figure 72 & Figure 73). The basic format of these functions is to take in a reference to the data being measured and the pixels-per-millimetre ratio, and then pass those values to their respective pixel measurement functions, and then perform the relevant conversion to return the result in millimetres.

```
    # Get the area of a contour in millimetres^2
    def get_contour_area_mm(contour, pixels_per_millimetre_ratio):
        # Get the area in pixels^2 first
        area_px = get_contour_area_px(contour)
        
        # Convert the result to millimetres^2
        area_mm = pixels_to_millimetres(area_px, pixels_per_millimetre_ratio, 2)
        
        return area_mm
```

Figure 71 - Function for Calculating Contour Area in Millimetres

```
# Get the X and Y dimensions of a countour in millimetres
def get_contour_size_mm(contour, pixels_per_millimetre_ratio):
    # Get the size in pixels first
    x_length_px, y_length_px = get_contour_size_px(contour)
    
    # Convert the result to millimetres
    x_length_mm = pixels_to_millimetres(x_length_px, pixels_per_millimetre_ratio)
    y_length_mm = pixels_to_millimetres(y_length_px, pixels_per_millimetre_ratio)
    
    return x_length_mm, y_length_mm
```

Figure 72 - Function for Calculating X and Y Lengths of a Contour in Millimetres

```
# Get the area of a circle in mm^2
def get_circle_area_mm(coin_area_px, pixels_per_millimetre_ratio):
    return pixels_to_millimetres(coin_area_px, pixels_per_millimetre_ratio, 2)
```

Figure 73 - Function for Calculating the Area of a Circle in Millimetres

```
# Get the radius of a circle in mm^2
def get_circle_radius_mm(radius_px, pixels_per_millimetre_ratio):
    return pixels_to_millimetres(radius_px, pixels_per_millimetre_ratio)
```

Figure 74 - Function for Calculating the Radius of a Circle in Millimetres

4.5 Testing Measurement Accuracy

Once the new measurement functions were complete, it was important to provide evidence that they were indeed more accurate than the previous measurement functions. To accomplish this, unit testing was performed on both the new measurement functions (Figure 75) as well as the old measurement functions (Figure 76) to test their respective accuracies.

The unit tests were developed using Python *unittest* library, and each test follows a similar structure:

1. Sample data is created. This data is the same for each test, and includes a coin radius in millimetres, a testing image, and an OpenCV circle extracted from the coin within the image.

2. If testing a contour, a testing contour is created. As the real-world sizes of wounds are not known, the tests instead create a copy of the coin circle and convert it to contour data to simulate a wound area with the same dimensions as a coin.

3. An ‘expected’ result is defined. As we are using coin dimensions for both circles and contours, the expected result will always be equal to either the radius, diameter or area of a coin.

4. The relevant measurement function is performed on the data and stored as the ‘result’.

5. The ‘result’ is then compared to the ‘expected’ value and an accuracy percentage is calculated based on the difference between what was expected and what the function returned.

6. The test returns either a ‘pass’ or ‘fail’ depending on if the ‘result’ matched the ‘expected’ value. A margin of error of 1% was applied as various factors, such as rounding errors are expected and perfect results are unlikely.

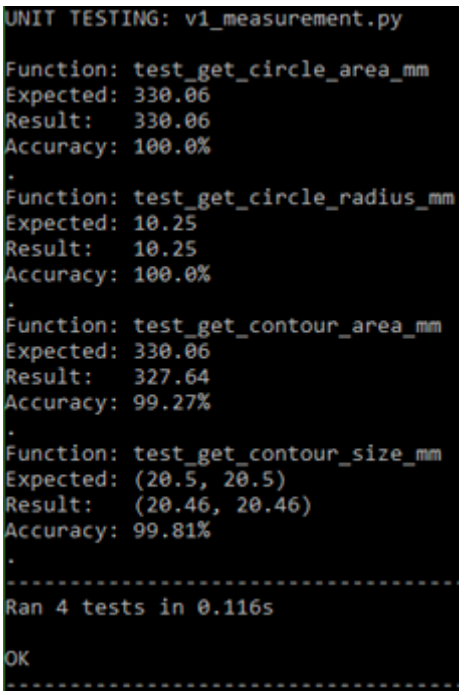

Figure 75 - Testing Accuracy New Measurement Functions

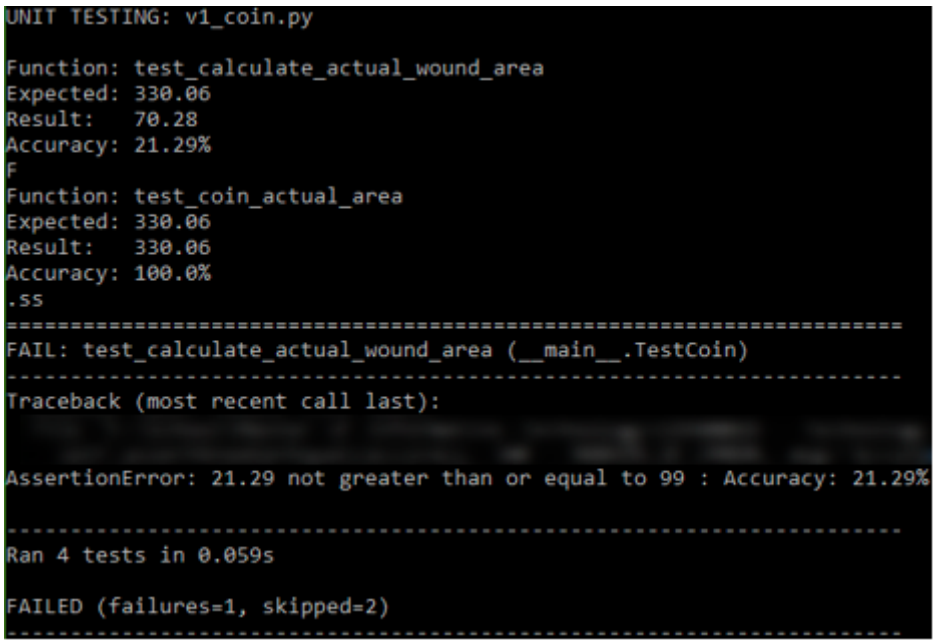

Figure 76 - Testing Accuracy of Old Measurement Functions

Unit testing was also performed to compare the accuracy of the old and new accuracies with each other. As the old measurement functions only included area measurements, the results of those tests were compared directly with each other.

The outcome of these unit tests demonstrate that while measuring coin areas were equally accurate, with both achieving an accuracy score of 100%, there is a significant improvement in accuracy when testing wound contour areas, where the old measurement function scored an accuracy of 21.29% and the new function scored an accuracy of 99.27%, as seen in Figure 77. This, combined with the addition of new measurements, which also prove to be highly accurate are a significant improvement over the measurement tools developed earlier in the project.

```
COMPARING TEST RESULTS

Function: test_circle_area_comparison
Winner:   DRAW (Accuracy: 100.0% - 100.0%)
Loser:    -
.
Function: test_contour_area_comparison
Winner:   get_contour_area_mm (Accuracy: 99.27%)
Loser:    calculate_actual_wound_area (Accuracy: 21.29%)
.
----------------------------------------------------------------------
Ran 2 tests in 0.059s
```

Figure 77 - Comparing Accuracies of New and Old Measurement Functions

4.6 Visualising Results

In addition to printing text results to the command prompt, it was decided that displaying images which display the results would be a valuable outcome for the project. Initially we display the original image provided to the program (Figure 78), and then we display images visualising the measurements performed on the image.

As noted in previous sections, there are two types of data: Wounds represented by OpenCV contours, and coins represented by OpenCV circles. There are also two types of measurements being performed on each of these objects: Area measurements, and length measurements. To facilitate this, four visualisation functions were developed to display each type of measurement for each type of object.

For area visualisations, as seen in Figure 79 below, the objects are highlighted and outlined in blue, with their measurements printed in the centre. If names are provided, such as the current project which assigns wounds a number to account for multiple wounds in a single image, they are also displayed inside the objects area above their measurement text. The function for visualising circle areas can be seen in Figure 82, and contours in Figure 83.

Length visualisation took a little more work to accomplish, and could benefit from further modification, but the current iteration, which can be seen in Figure 80, displays the areas being measured which, as described in the measurement section, is a circle for the coin, and a bounding box for the wound. This was done to help highlight exactly how the measurements are being calculated, particularly for wound

lengths, which clearly displays that the bounding box represents the maximum lengths of each axis.

Then, lines are drawn across the length of the object, once for circles, and one for each axis of a contour, with their measurements printed at the end of each line. Similar to the area visualisations, a name is also displayed in the centre of the object if one is provided. The code for visualising circle lengths can be see in Figure 84, and contours in Figure 85.

A function for visualising the radius of a circle was also developed but is not used in the current. The code for this, as well as various helper functions named in some of the images provided, can be seen in the GitHub repository.

For testing purposes, the binary mask generated for the wound can also be displayed, as seen in Figure 81.

The resulting visualisations should help give at-a-glance recognition of the measurements performed and assist in viewing and understanding the analysis being performed on the images. I'm not completely satisfied with how wound lengths are being displayed, as while the objective was to help show how the measurements are being performed, I feel it is a bit too 'noisy' and might reduce readability. In a future iteration I would work on a simplified, more streamlined visualisation for this.

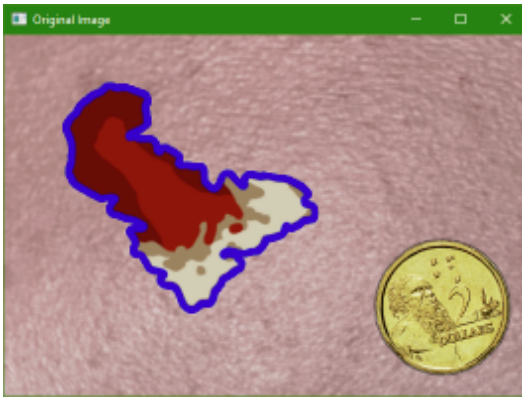

Figure 78 - Visualising Original Image

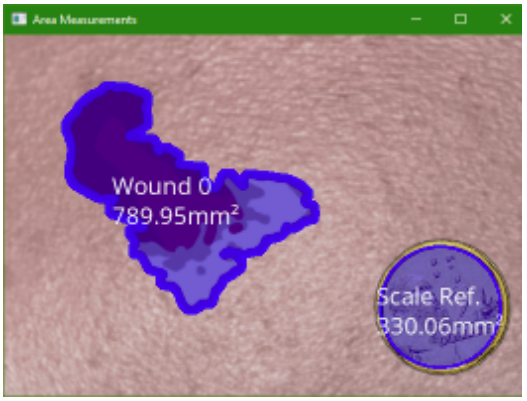

Figure 79 - Visualising Area Measurements

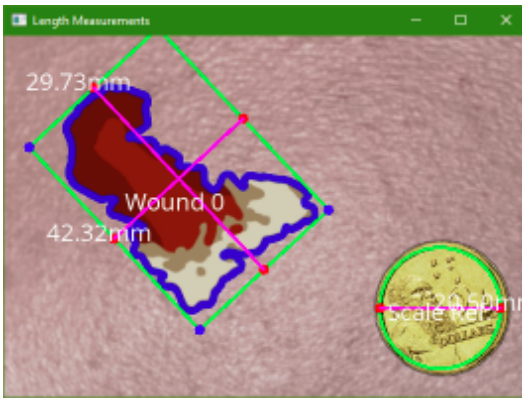

Figure 80 - Visualising Length Measurements

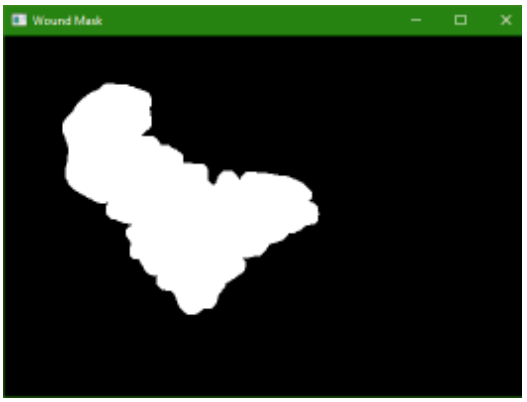

Figure 81 - Visualising Wound Mask

```python
# Visualise a OpenCV circle showing its area in millimetres
def visualise_circle_area_mm(image, circle, area_mm, name=None):
    position_x, position_y, radius_px = circle

    # Create a copy of the image
    overlay_image = image.copy()

    # Draw a cirle on the overlay image
    cv2.circle(overlay_image, (position_x, position_y), radius_px, (255, 0, 0), -1)

    # Overlay the image on the original using alpha transparency
    alpha = 0.5
    image = cv2.addWeighted(image, 1 - alpha, overlay_image, alpha, 0)

    # Draw a ring around the circle
    cv2.circle(image, (position_x, position_y), radius_px, (255, 0, 0), 2)

    # Add the area measurement as text
    text = f'{area_mm:.2f}mm²'

    if name:
        text = str(name) + '\n' + text

    image = drawText(image, text, position_x, position_y)

    return image
```

Figure 82 - Function for Visualising the Area of a Circle

```python
# Visualise an OpenCV contour showing its area in millimetres
def visualise_contour_area_mm(image, contour, area_mm, name=None):
    # Create a copy of the image
    overlay_image = image.copy()

    # Draw the contour on the overlay image
    cv2.drawContours(overlay_image, [contour], -1, (255, 0, 0), -1)

    # Overlay the image on the original using alpha transparency
    alpha = 0.5
    image = cv2.addWeighted(image, 1 - alpha, overlay_image, alpha, 0)

    # Draw the outline of the contour
    cv2.drawContours(image, [contour], -1, (255, 0, 0), 2)

    # Get the centre point of the contour
    moments = cv2.moments(contour)
    center_x = int(moments['m10'] / moments['m00'])
    center_y = int(moments['m01'] / moments['m00'])

    # Add the area measurement as text
    text = f'{area_mm:.2f}mm²'

    if name:
        text = str(name) + '\n' + text

    image = drawText(image, text, center_x, center_y)

    return image
```

Figure 83 - Function for Visualising the Area of a Contour

```python
# Visualise a OpenCV circle showing its diameter in millimetres
def visualise_circle_diameter_mm(image, circle, diameter_mm, name=None):
    position_x, position_y, radius_px = circle

    # Draw a ring around the circle
    cv2.circle(image, (position_x, position_y), radius_px, (0, 255, 0), 2)

    # Draw a dot at each edge
    cv2.circle(image, (position_x - radius_px, position_y), 5, (0, 0, 255), -1)
    cv2.circle(image, (position_x + radius_px, position_y), 5, (0, 0, 255), -1)

    # Draw a line through the center of the circle
    cv2.line(image, (position_x - radius_px, position_y),
                    (position_x + radius_px, position_y), (255, 0, 255), 2)

    # Add the wound name as text, if one is given
    if name:
        image = drawText(image, name, position_x, position_y)

    # Add the radius measurement as text
    image = drawText(image, f'{diameter_mm:.2f}mm', position_x + radius_px - 15, position_y - 10)

    return image
```

Figure 84 - Function for Visualising the Diameter of a Circle

```
# Visualise an OpenCV contour showing width and height in millimetres
def visualise_contour_size_mm(image, contour, size_x_mm, size_y_mm, name=None):
    # Calculate a rect around the contour.
    rect = cv2.minAreaRect(contour)

    # Covert the rect to a list of points
    box = cv2.boxPoints(rect)
    box = np.int0(box)

    # Draw the box around the contour
    cv2.drawContours(image, [box], 0, (0, 255, 0), 2)

    # Get the corners of the box
    top_left, top_right, bottom_right, bottom_left = box

    # Draw a circle at each corner
    cv2.circle(image, top_left, 5, (255, 0, 0), -1)
    cv2.circle(image, top_right, 5, (255, 0, 0), -1)
    cv2.circle(image, bottom_right, 5, (255, 0, 0), -1)
    cv2.circle(image, bottom_left, 5, (255, 0, 0), -1)

    # Calculate the midpoints for each edge
    top_midpoint = get_midpoint(top_left, top_right)
    bottom_midpoint = get_midpoint(bottom_left, bottom_right)
    left_midpoint = get_midpoint(top_left, bottom_left)
    right_midpoint = get_midpoint(top_right, bottom_right)

    # Draw circles at middle of each edge
    cv2.circle(image, top_midpoint, 5, (0, 0, 255), -1)
    cv2.circle(image, bottom_midpoint, 5, (0, 0, 255), -1)
    cv2.circle(image, left_midpoint, 5, (0, 0, 255), -1)
    cv2.circle(image, right_midpoint, 5, (0, 0, 255), -1)

    # Draw lines between midpoint circles
    cv2.line(image, top_midpoint, bottom_midpoint, (255, 0, 255), 2)
    cv2.line(image, left_midpoint, right_midpoint, (255, 0, 255), 2)


    # Add the wound name as text, if one is given
    if name:
        # Get the centre point of the contour
        moments = cv2.moments(contour)
        center_x = int(moments['m10'] / moments['m00'])
        center_y = int(moments['m01'] / moments['m00'])

        image = drawText(image, name, center_x, center_y)

    # Add the size measurements as text
    image = drawText(image, f'{size_x_mm:.2f}mm', top_midpoint[0] - 15, top_midpoint[1] - 10)
    image = drawText(image, f'{size_y_mm:.2f}mm', left_midpoint[0] - 15, left_midpoint[1] - 10)

    return image
```

Figure 85 - Function for Visualising the X and Y Lengths of a Contour

4.7 Saving, Loading and Comparing Results

Comparing results of wound analysis over time was originally given as one of the core outcomes for the project, however after the first meeting with the client it was scoped out. However, in the final few days of development I found myself with some free time, so I decided to develop a quick and simple prototype to demonstrate how such a system could work.

For this prototype, the resulting data from wound analysis can be saved to file and loaded later for comparison. While a final solution would likely make use of a database system, for this prototype I opted for a simple JSON data structure which is saved directly to the project folder. It stores a name, which is used to compare results of the name value, the date of processing, links to image files which are also saved alongside the JSON file, and the results of image analysis. The process for this can be seen in Figure 86, and an example of the JSON data it creates can be seen in Figure 87.

```
def save_wound_data(output_path, name, image, mask, wound_results):
    # Check if any wound results exist, otherwise there is nothing to save
    if wound_results:
        # Create folders
        output_subpath = os.path.join(output_path, name)
        os.makedirs(output_subpath, exist_ok=True)

        # Save the current date/time
        filename_timestamp = datetime.now().strftime('%Y-%m-%d_%H-%M-%S') # Filename-friendly format

        # Save the image
        image_filename = f'wound_image_{name}_{filename_timestamp}.jpg'
        image_path = os.path.join(output_subpath, image_filename)
        cv2.imwrite(image_path, image)

        # Save the mask
        mask_filename = f'wound_mask_{name}_{filename_timestamp}.jpg'
        mask_path = os.path.join(output_subpath, mask_filename)
        cv2.imwrite(mask_path, mask)

        # Format the wound data as JSON
        data = {}
        data['name'] = name
        data['date'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S') # More readable format

        data['image'] = image_path
        data['mask'] = mask_path
        data['wounds'] = []

        # Loop through each contour and create sub nodes for each
        for i, result in enumerate(wound_results):
            wound = {}
            wound['id'] = i
            wound['size_x'] = result[0]
            wound['size_y'] = result[1]
            wound['area'] = result[2]

            data['wounds'].append(wound)

        # Save the JSON file
        json_filename = f'wound_result_{name}_{filename_timestamp}.json'

        with open(os.path.join(output_subpath, json_filename), 'w') as file:
            json.dump(data, file)

        print()
        print(f'Results Saved: "...\{output_subpath}"')
    else:
        print('No countours found in', contours)
```

Figure 86 - Function for Saving Wound Data

```
{
    "name":"heal_test",
    "date":"2023-05-14 21:52:33",
    "image":"results\\heal_test\\wound_image_heal_test_2023-05-14_21-52-33.jpg",
    "mask":"results\\heal_test\\wound_mask_heal_test_2023-05-14_21-52-33.jpg",
    "wounds":[
        {
            "id":0,
            "size_x":39.14118097140916,
            "size_y":48.56263825257012,
            "area":1296.72141225477
        }
    ]
}
```


Figure 87 - Example of Saved JSON Data Structure

Saved data can then be loaded, and loading multiple results can then be compared to display progress of a wound over time. The process of loading data can be seen in Figure 88.

```
def load_wound_data(path):
    # Check to see if path given has any results
    if os.path.exists(path) and os.listdir(path):
        print()
        print('Loading results...')
        # Load the saved data
        results = []

        # Loop through each file in the folder
        for file_name in os.listdir(path):
            file_path = os.path.join(path, file_name)

            # Check if the file is a JSON file
            if file_name.endswith('.json'):
                # Add it to the list
                with open(file_path) as file:
                    results.append(json.load(file))

        # If any results were loaded, return the list
        if results:
            return results

        # Otherwise print an error and return None
        print()
        print(f'No results exist at "...\{path}"')

        return None

    else:
        print(f'No results exist at "...\{path}"')
```

Figure 88 - Function for Loading Wound Data

Currently, the comparison takes the form of a line graph (Figure 89), which plots the changes in area, and lengths of wounds over time. There is also some code existing to compare colour data, however due to time constraints this was not fully integrated into the current comparison system.

Figure 89 - Line Graph Plotting Wound Measurements Over Time

5. Colour analysis

5.1 Solutions Researched

During the development of the concept, the research for the colour analysis aspect was mainly directed towards the article by (Runjie B. Shi, Jimmy Qiu and Vincent Maida, 2019). The design aspect involved transforming the image into a Hue-Saturation-Value space where hues are the RGB colours, the saturation depicts the potency of a colour and how bright or dark a colour is was presented by the value. The wound image would then go through colour transformations in the Black-Yellow-Red model that involves oversaturating the image to these specific colours to extract the colour information.

$$\%\text{granulation} = \frac{\text{number of “granulation” pixels}}{\text{total number of pixels in wound}}$$

$$\%\text{epithelial} = \frac{\text{number of “epithelial” pixels}}{\text{total number of pixels in wound}}$$

The main chronic wound types presented by Runjie B. Shi are Granulation, Epithelial and Slough. During implementation, one of the core issues being encountered was that black values would become very troublesome to extract and group into specific wound types as the model could not decide at what stage was the necrotic tissue positioned as necrotic tissues can arrive in many colours and hues such as dark greys, blacks, browns and dark yellows, this is evident in the research itself as it does not contain any images with necrotic tissue.

Chronic Wound Stages:

1. Necrotic: Dead tissue that is present on the body that needs to be identified and dealt with immediately.
2. Slough: Depicts tissue that is yellow or off-white in colour and is mainly a thick liquid puss that is prone to infections.
3. Granulation: Shows new tissue and blood vessels being created, depicted in bright reds and pink colours.
4. Epithelial: Is the final stage of healing and is represented by epidermis tissue on the wound surface that is light pink and becomes stronger over time.

Although this model addresses the many different lighting conditions present in the wound images, the information about the wound that is being presented was geared more towards patient self-care instead of telehealth and did not provide the necessary information that a doctor might be looking for during appointments. When it comes to histogram depictions a doctor would also need to spend a considerable amount of time trying to understand exactly what information is being presented which is a detriment to nurses and doctors who may be dealing with multiple patients and appointments and running short on time.

5.2 HSV Iteration

The first iteration of the colour analysis aspect was mainly inspired by the research done by (Runjie B. Shi, Jimmy Qiu and Vincent Maida, 2019). The colour extraction model involves successfully applying thresholding and performing segmentation on the wound using the Black-Yellow-Red model in the Hue-Saturation-Value colour space, The categorisation would then be honed via morphological transformations.

Another advantage and one of the main reasons this approach was chosen was due to its resistance to different lighting conditions and how robust it would be when handling pictures from various kinds of smartphones.

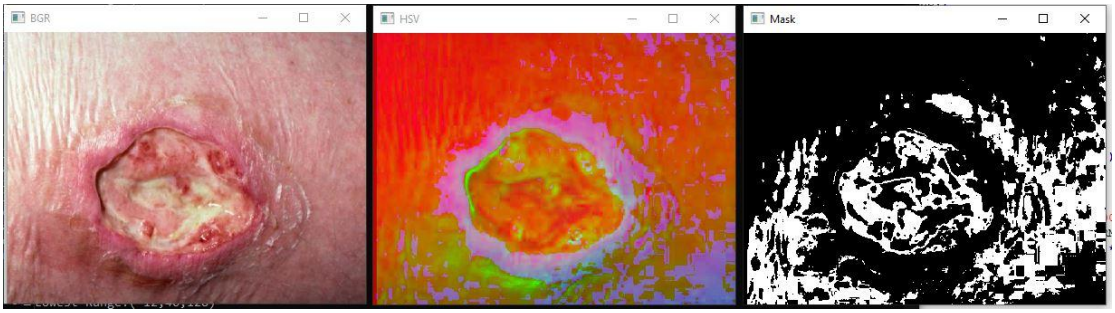

Figure 90: BGR, HSV and Masking Results for colour extraction.

The first implementation accepted an image, transformed it in the HSV space and also created a mask for viewing purposes. The user would then be able to select/click inside the HSV image to understand exactly how the wound is progressing. Selecting specific aspects of the wound would change the histogram to show exactly what aspects of the wound have changed compared to previous wound images. At certain points if the colours breach the percentage barrier it would also mean that there has been a drastic change from what was given previously.

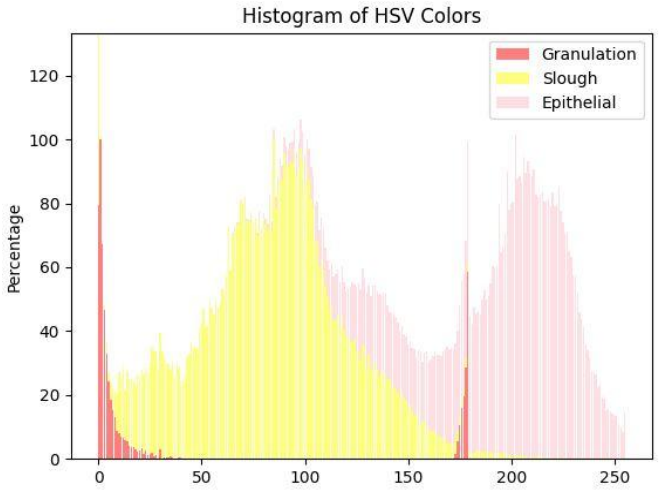

Figure 91: Histogram Results for colour extraction.

After careful consideration, and by also taking some feedback from the doctor and our client, some major design changes were made to have the colour extraction approach geared towards telehealth. This involved replacing the HSV method with a more conventional colour extraction technique.

5.3 Image Preparation

To create consistency between the main project and also account for the needs of the doctor, the decision was made to implement an orthodox and easy to understand method for colour extraction while also borrowing some elements from the previous iteration. The first step involved creating a simple function that resizes and transforms the image and mask for accurate and hastened colour extraction. In the case of colour extraction, resizing the image does not yield any negative effects, but is essential due to the fact that having too high a pixel count could result in the computation time being abnormally long . The next step is to replace the black values of the mask before colour extraction as we would need access to black values to calculate the amount of necrotic tissue in the wound. Those values can then be excluded during colour extraction.

```python
#Processes the wound and mask image
def process_image(image_path, mask_path, resize_shape=(400, 300), kernel_size=(10, 10), num_clusters=4):
    # load images and masks
    image = cv2.imread(image_path)
    mask = cv2.imread(mask_path, 0)

    # convert images and masks to RGB
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # resize images and masks
    image_resized = cv2.resize(image, resize_shape)
    mask_resized = cv2.resize(mask, resize_shape)

    # replace black values in mask
    mask_replaced = replace_black_value(mask_resized)

    # clean the mask of artifacts and noise
    kernel = np.ones(kernel_size, np.uint8)
    cleaned_mask = cv2.morphologyEx(mask_replaced, cv2.MORPH_CLOSE, kernel)

    # extract color information from masks within white areas
    white_pixels = np.where(cleaned_mask == 255)
    masked_image = image_resized[white_pixels]

    # colour quantization to reduce colors
    quantized_image, centers = quantize_image(masked_image, num_clusters)
    cv2.imshow('mask', mask_replaced)
    cv2.imshow('cleaned', cleaned_mask)

    return image_resized, cleaned_mask, quantized_image, centers
```

Figure 92: depicts the process_image function.

Figure 93: coloured and cleaned mask from project files.

A particular problem that we encountered was the amount of artefacts and noise that resided in the mask. The solution to cleaning the mask came from morphological transformations and setting kernel sizes that allowed us to remove unwanted noise. The image can then be quantized for palette simplification and to create some consistency between the colours. Another crucial step is to exclude a certain threshold of colours that the doctor might use to draw the border of the wound so that it doesn't interfere with the calculation. The colours of the image have also been normalised in a LAB colour space to account for various lighting conditions.

5.4 Colour Extraction and Categories

The colours of the wound can then be extracted after the appropriate mask has been applied. The colours can then be plotted on a pie chart to aid easy viewing and quick understanding for the doctor. The amount of colours that are being collected can be enhanced by changing the number of clusters present in the code, allowing for more detailed extraction (4 is the optimal value).

```
# calculates the percentage of colors in image
def calculate_color_percentage(image, exclude_color= np.array([24, 23, 172]), distance_threshold=50)
    unique_colors, counts = np.unique(image.reshape(-1, 3), axis=0, return_counts=True)
    color_percentages = []

    for color, count in zip(unique_colors, counts):
        color_hex = "#{:02x}{:02x}{:02x}".format(color[0], color[1], color[2])
        if exclude_color is not None:
            color_distance = np.linalg.norm(color - exclude_color)
            if color_distance <= distance_threshold:
                continue
        percentage = (count / np.sum(counts)) * 100
        color_percentages.append((color_hex, percentage))

    return color_percentages
```

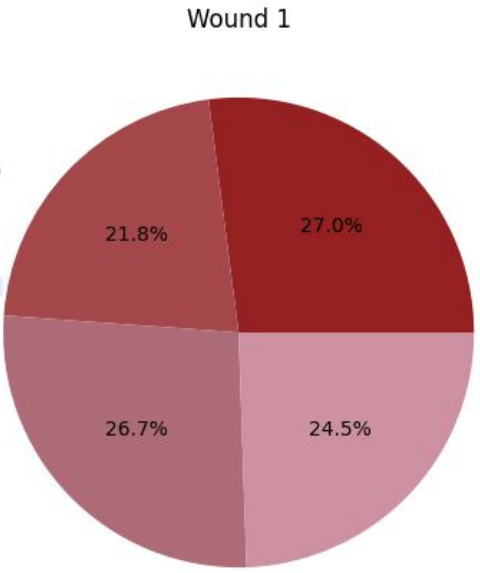

Figure 94: calculate_colour_percentages function and pie chart of wound colours.

The next step is to create a colour dictionary that is referenced to group all of the colours present on the wound into chronic wound types that will allow the doctor to understand exactly what type of wound it is. The grouping is determined by a similarity thresholding parameter that can be lowered for stricter groupings or raised to allow for more colours. The distance between the colours from the wound to the colours inside the dictionary is what determines if the colours will be allowed into the wound category.

```
group_dictionary = {
    'Slough': np.array([255, 255, 0]),
    'Granulation': np.array([187, 11, 13]),
    'Epithelial': np.array([240, 133, 96]),
    'Necrotic (Late)': np.array([0, 0, 0]),
    'Necrotic (Early)': np.array([165, 42, 42])
}
```

Figure 95: wound dictionary depicting an array of RGB colour values for thresholding.

```
# groups colors into categories
def group_colors(color_percentages, color_similarity_threshold=200):
    color_groups = {}

    for color, percentage in color_percentages:
        color_rgb = np.array([int(color[1:3], 16), int(color[3:5], 16), int(color[5:7], 16)])

        for group_name, group_color in wound_dictionary.items():
            dist = calculate_distance(color_rgb, group_color)
            if dist < color_similarity_threshold:
                if group_name in color_groups:
                    color_groups[group_name] += percentage
                else:
                    color_groups[group_name] = percentage
                break

    for group_name in wound_dictionary.keys():
        if group_name not in color_groups:
            color_groups[group_name] = 0

    grouped_color_groups = [(group_name, percentage) for group_name, percentage in color_groups.items()]
    return grouped_color_groups
```

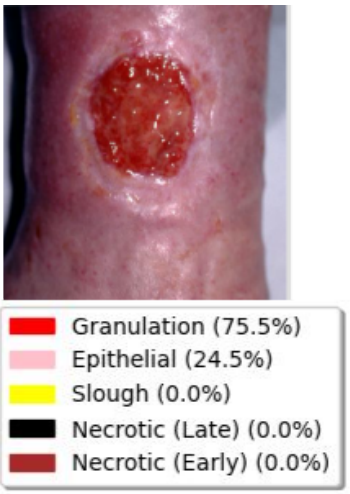

Figure 96: group_colours function and wound categories with percentages.

5.5 Colour Comparison and Analysis

Additionally, the solution has to show a comparison of the colour groups when compared to the previous image. This function is crucial as it will allow the doctor to make an accurate analysis on how the wound is progressing. When certain colours or groups disappear or evolve, it will be denoted with a “+” or “-” to signify change.

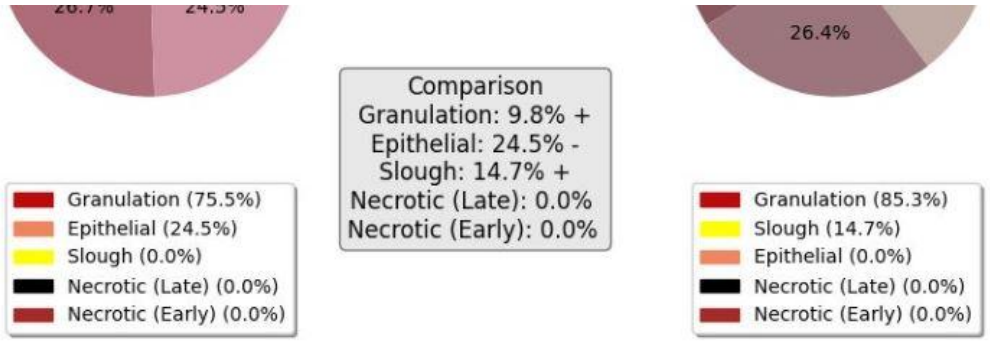

Figure 97: group_colours function and wound categories with percentages.

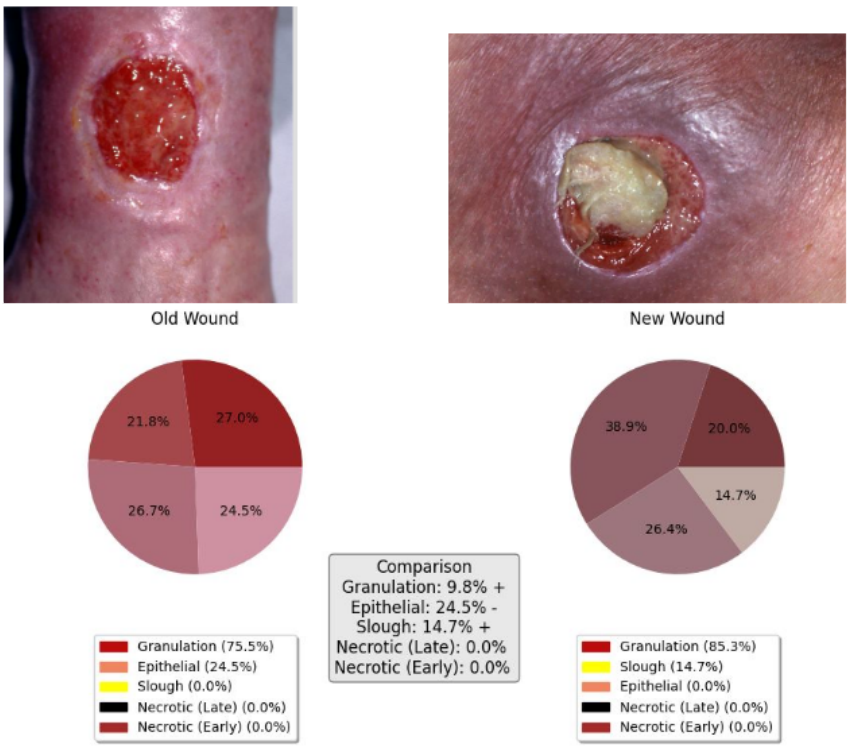

Figure 98: Comparison of two wounds in the ulcer_samples file provided by the client.

Finally, The image above displays the final product of the colour analysis aspect of the project. The wound images are displayed, the colour analysis is depicted in the form of a pie chart. The chronic wound categories contain groupings of the colours that were extracted and a comparison is made between wounds to show if the progression of the wound is advantageous or detrimental to the patients' health.

6. Conclusion and future work

In conclusion, the deliverables and goals set by our clients were met by developing a centralised solution that houses the ability to detect the border of the wound in multiple different ways by utilising a deep learning model. A great amount of agency has also been provided to the client/doctor in case they would like to create the borders and analyse the wound themselves. The ability to measure the size of the wound was also accounted for with the implementation of the coin detection feature and creating comparisons with the wound to receive accurate measurements of the wound bed. By using the image processing techniques, we were also able to obtain the accurate colours that are found within these chronic ulcer wounds provided by the client and those colours have also been categorised to provide an accurate description of the wound for any doctor or nurse that would engage with this software. Additionally, the team were also able to implement comparisons between fresh and

old wounds, which was initially considered to be out of scope by displaying a line graph showing the size of the wound either increasing or decreasing over a period of time while also having a comparison label showcasing how a wound category is evolving. The research that was conducted during the concept development phase has been extremely beneficial as a plethora of our analysis aspects have been developed around the important information that was extracted from the many articles and research papers.

As for the further work involved, currently the colour analysis isn't fully integrated with the main program, While all of the code has been refactored for use within the main file (and can be found within Github), it has not been fully incorporated into Version_1.py and the colour analysis file needs to be run separately to obtain accurate results.

7. References

Farley, P. (2023) What are Azure Cognitive Services? - azure cognitive services, Azure Cognitive Services | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/azure/cognitive-services/what-are-cognitive-services (Accessed: April 12, 2023).

Farley, P. (2023) What is image analysis? - azure cognitive services, Azure Cognitive Services | Microsoft Learn. Available at: https://learn.microsoft.com/en-us/azure/cognitive-services/computer-vision/overview-image-analysis?tabs=4-0 (Accessed: April 13, 2023).

Google search policies for Images & Video Boxes (2023) Google Search Help. Available at: https://support.google.com/websearch/answer/10688017?hl=en (Accessed: 24 May 2023).

Mishra, A. (2019) Machine learning in the AWS cloud: Add intelligence to applications with Amazon Sagemaker and Amazon Rekognition, Amazon. Sybex, a Wiley brand. Available at: https://aws.amazon.com/rekognition/custom-labels-features/ (Accessed: April 12, 2023).

Mishra, A. (2019) Machine learning in the AWS cloud: Add intelligence to applications with Amazon Sagemaker and Amazon Rekognition, Amazon. Sybex, a Wiley brand. Available at: https://docs.aws.amazon.com/rekognition/latest/dg/what-is.html (Accessed: April 12, 2023).

Rosebrock, A 2021, “Measuring size of objects in an image with opencv,” PyImageSearch, viewed 2 April, 2023, https://pyimagesearch.com/2016/03/28/measuring-size-of-objects-in-an-image-with-opencv/

Shi, R. B., Qiu, J., & Maida, V. (2019). Towards algorithm-enabled home wound monitoring with smartphone photography: A hue-saturation-value colour space thresholding technique for wound content tracking. International wound journal, 16(1), 211–218. https://doi.org/10.1111/iwj.13011

Venkatesh, B. (2021) How does the machine read images and use them in computer vision?, Top Website Designers, Developers, Freelancers for Your Next Project. Available at: https://www.topcoder.com/thrive/articles/how-does-the-machine-read-images-and-use-them-in-computer-vision (Accessed: April 13, 2023).

Stock photo, royalty-free image prices and plans (2023) Shutterstock. Available at: https://www.shutterstock.com/pricing (Accessed: 24 May 2023).

8. Appendix

Team Contribution Breakdown

Team member	Contribution
Lucas Qin	Executive Summary Section 1.1 Overview Section 1.2 Project background Section 2 Border detection References, Editing, Document Formatting
Hariesh Kumar Ravichandran	Section 1.3 Project Scope & Objectives Section 5 Colour Analysis Section 6 Conclusion and Further work References Document Editing & Formatting.
Andrew Oates	Section 4 Improving Wound Measurements Document Editing & Formatting
Natalie Huang	Section 3 Coin detection and Wound measurement Document formatting
Mona Ma	Section 1.3 Scope & Objectives Section 1.4 Project deliverables Section 2 Border detection References Editing, Formatting, Document Setup

Project Link in GitHub

https://github.com/lucas-project/Woud_Detection_Machine_Learning