Table of Contents

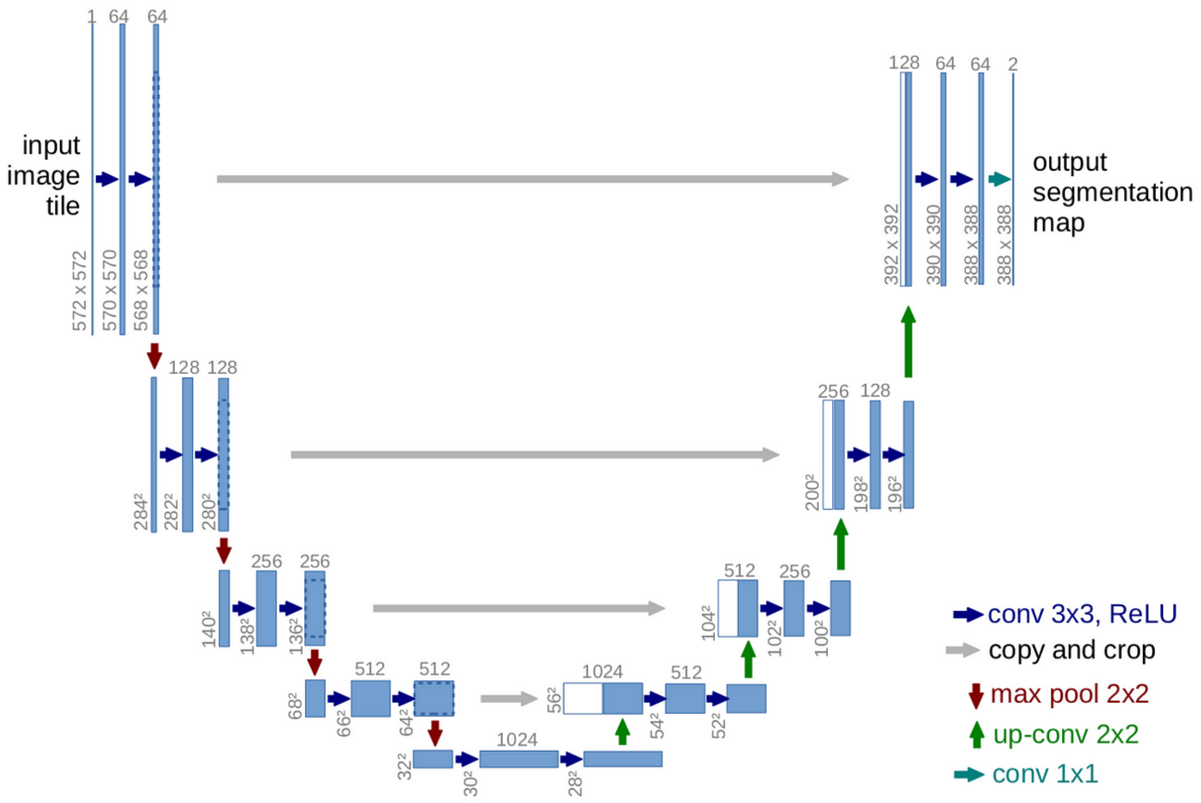
Methodology

Data Collection and Preparation

The dataset contain 1000 images about medical, collected from Kaggle website. To increase the number of images, we have to perform data augmentation to the original dataset and the details can be found in implementation section. The data included medical images and the targeted image (we call it mask in the later section). The size of the images is from 332 x 487 to 1920 x 1071, to make the process systematically, we have resized all the images into size 256 x 256.

Description of Architecture

The U-net architecture has been developed for image segmentation task specially for medical image. (Ronneberger et al., 2015) In this experiment, we are going to build this architecture using python Keras library. The figure below shows the architecture of U-net.



Note that in the original U-net architecture shown above, the input image has size 572 x 572. In this experiment, we are going to resize the image to 256 x 256, another algorithm remains the same with original.

The details of this architecture will show in the table below.

Model Details

Layer	Number of Filters	Number of Convolutions	Concatenate
1	64	2	Yes
2	128	2	Yes
3	256	2	Yes
4	512	2	Yes
Bridge	1024	2	No
Output	1	NA	NA

Convolutional Block Details

Operation	Kernel Size	Activation	Kernel Initializer	Padding
Convolution	3 x 3	relu	he_normal	same

Encoder Block Details

Step	Description
1	Convolution
2	Max Pooling with kernel size (2 x 2)

Decoder Block Details

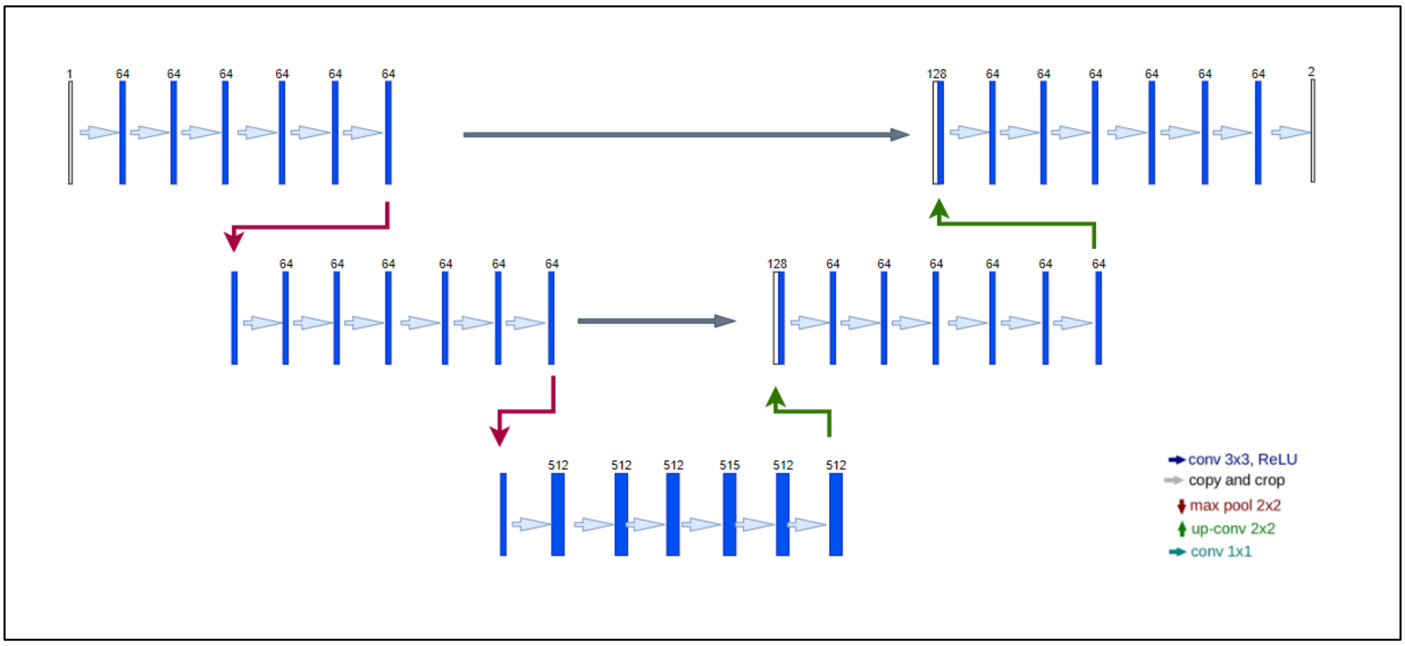
Step	Description
1	Conv2D Transpose Kernel size: (2 x 2) Strides: (2 x 2) Padding: same
2	Concatenate – Combining the output from previous block with the output of the encoder block in the same layer.
3	Convolution

Modification of Architecture

In this section, we are going to modify the architecture of original U-Net and observing the model performance. The modification including the change of convolutional layer, number of filters, layer of encoder and decoder blocks, and more.

Modification 1 (U1)

The first modification will be increasing the number of convolutions operation and decrease the filters in encoder and decoder blocks and bridge. The remains will be same with the original U-net. The architecture diagram is show below.



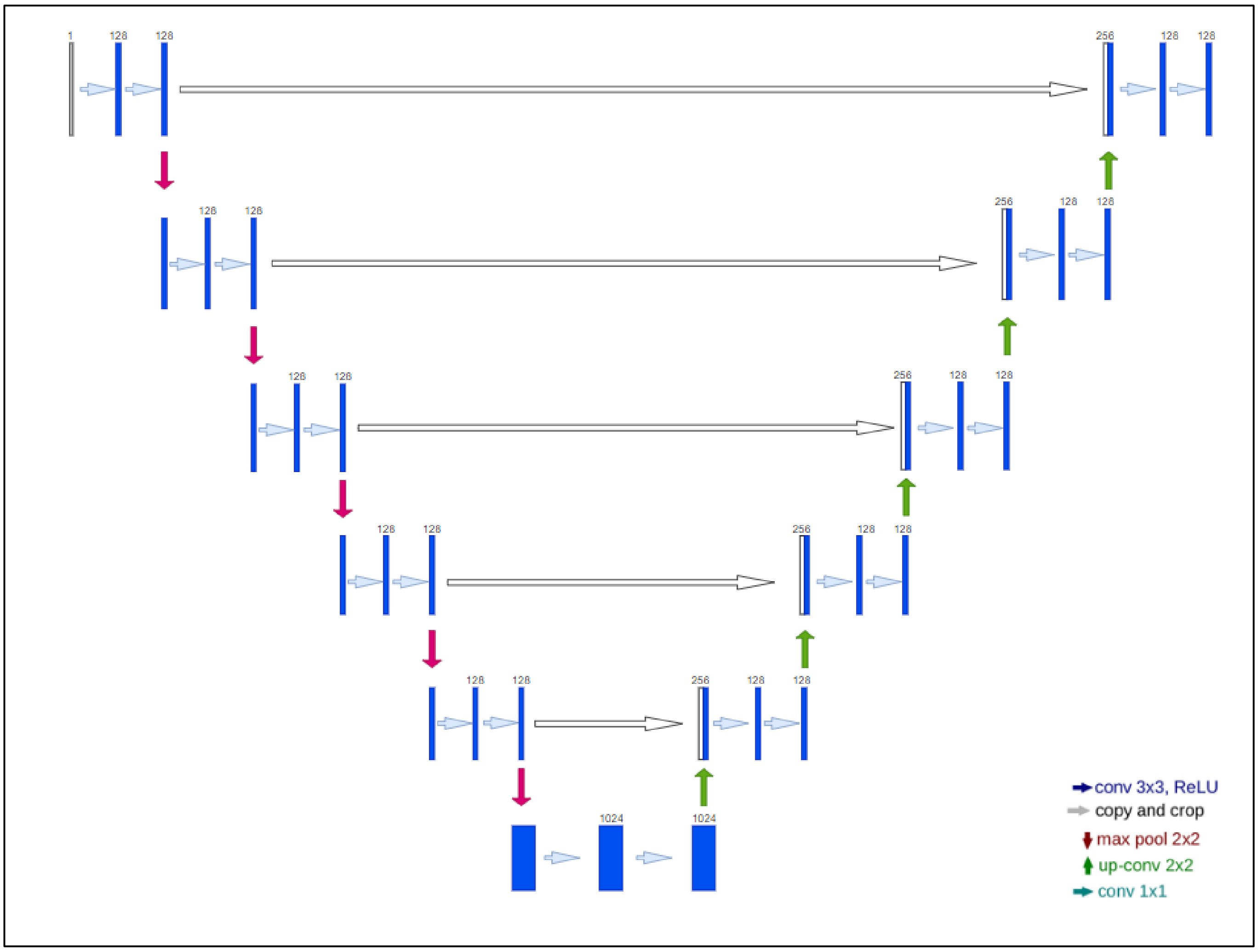
From the diagram above, we can see that the modified architecture has 2 encoder and decoder layers, but the number of convolution operations in increased.

Model Details

Layer	Number of Filters	Number of Convolutions	Concatenate
1	64	7	Yes
2	128	7	Yes
Bridge	1024	7	No
Output	1	NA	NA

<u>Modification 2 (U2)</u>

Similarly, the second modification will be increasing the number of encoder and decoder layer, and we have set a fixed value for the number of filters for each encoder and decoder layer. Here is the architecture diagram for this modification.



<u>Model Details</u>

Layer	Number of Filters	Number of Convolutions	Concatenate
1	128	2	Yes
2	128	2	Yes
3	128	2	Yes
4	128	2	Yes
5	128	2	Yes
Bridge	1024	2	No
Output	1	NA	NA

Evaluation Metric

In this experiment, the evaluation metric to evaluate the model performance will be accuracy and intersection over union (IoU).

Accuracy

Accuracy is a metric that commonly use for predicts non-continuous numeric output, this metric measures the proportion of true predictions given by the model. The formula for computing accuracy is show below.

$$\boxed{Accuracy = \frac{TP + TN}{TP + TN + FP + FN}}$$

where TP denotes true positive, TN denotes true negative, FP denotes false positive, and FN denotes false negative.

Intersection over Union (IoU)

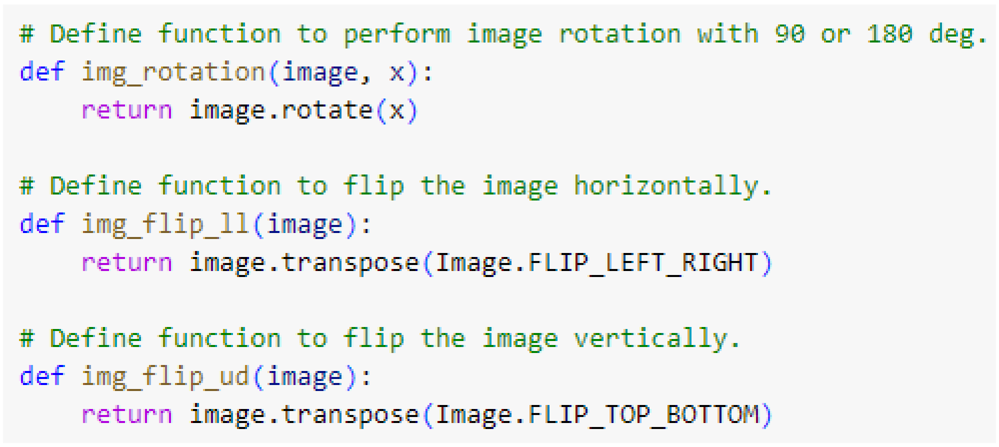
Intersection over union IoU is a metric used for image segmentation task. This metric measures the area between the prediction and the truth boundary. This formula computes the area of intersection between that two boundary and divide the union of that two boundary. The mathematical expression will be

$$\boxed{IoU = \frac{Area\ of\ Intersection}{Area\ of\ Union} = \frac{|A \cap B|}{|A \cup B|}}$$

where A denotes the region of prediction given by the model, and B denotes the truth region of the input image.

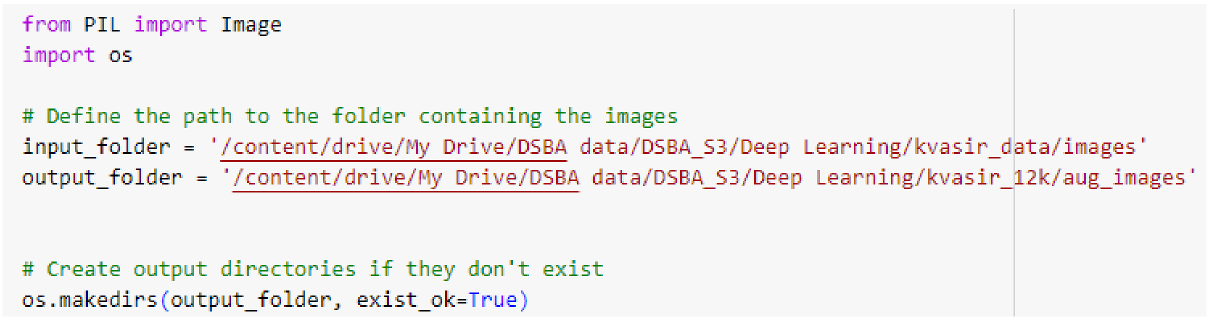
Data Augmentation

The dataset collects from Kaggle have 1000 of images. To increase the dataset for model development, we need more data. So, we are going to perform data augmentation to generates more images from the original images. For each image, we are going to flips horizontally and vertically, then rotation for 90°, 180°, and 270°. So, there are total $3 \times 4 = 12$ permutations for each original images. Hence, the dataset will become 12,000 images.



```
# Define function to perform image rotation with 90 or 180 deg.
def img_rotation(image, x):
    return image.rotate(x)

# Define function to flip the image horizontally.
def img_flip_ll(image):
    return image.transpose(Image.FLIP_LEFT_RIGHT)

# Define function to flip the image vertically.
def img_flip_ud(image):
    return image.transpose(Image.FLIP_TOP_BOTTOM)
```

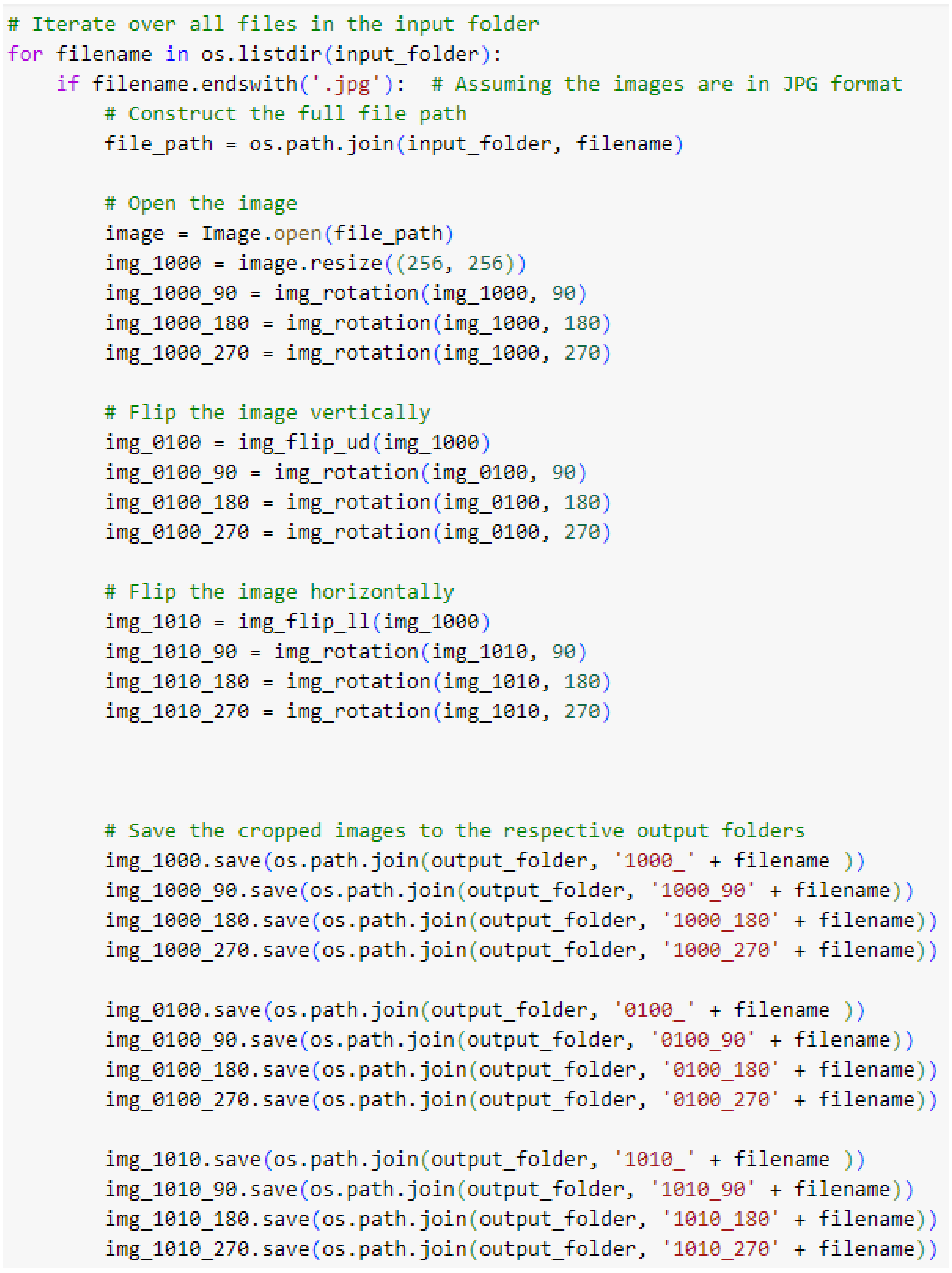
The image above is defining the function to perform data augmentation. Three functions have been defined, they are function to rotate the image, and flip the image by horizontally and vertically. Now, we are going to create the folder to store the augmented data.



```
from PIL import Image
import os

# Define the path to the folder containing the images
input_folder = '/content/drive/My Drive/DSBA data/DSBA_S3/Deep Learning/kvasir_data/images'
output_folder = '/content/drive/My Drive/DSBA data/DSBA_S3/Deep Learning/kvasir_12k/aug_images'


# Create output directories if they don't exist
os.makedirs(output_folder, exist_ok=True)
```

WLOG, the folder name ‘aug_images’ is the folder to store the augmented data for training. The augmented mask data will be store in another prepared folder.

To augment the image data, we have to run those three functions that we defined above for each of the original image.

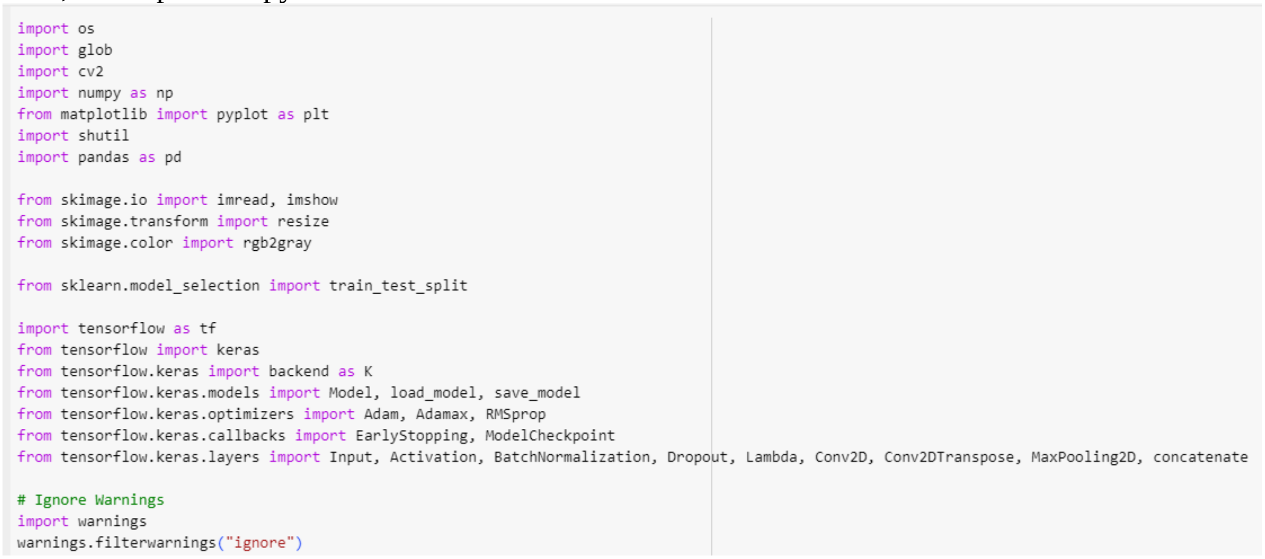


```python
# Iterate over all files in the input folder
for filename in os.listdir(input_folder):
    if filename.endswith('.jpg'):  # Assuming the images are in JPG format
        # Construct the full file path
        file_path = os.path.join(input_folder, filename)

        # Open the image
        image = Image.open(file_path)
        img_1000 = image.resize((256, 256))
        img_1000_90 = img_rotation(img_1000, 90)
        img_1000_180 = img_rotation(img_1000, 180)
        img_1000_270 = img_rotation(img_1000, 270)

        # Flip the image vertically
        img_0100 = img_flip_ud(img_1000)
        img_0100_90 = img_rotation(img_0100, 90)
        img_0100_180 = img_rotation(img_0100, 180)
        img_0100_270 = img_rotation(img_0100, 270)

        # Flip the image horizontally
        img_1010 = img_flip_ll(img_1000)
        img_1010_90 = img_rotation(img_1010, 90)
        img_1010_180 = img_rotation(img_1010, 180)
        img_1010_270 = img_rotation(img_1010, 270)



        # Save the cropped images to the respective output folders
        img_1000.save(os.path.join(output_folder, '1000_' + filename ))
        img_1000_90.save(os.path.join(output_folder, '1000_90' + filename))
        img_1000_180.save(os.path.join(output_folder, '1000_180' + filename))
        img_1000_270.save(os.path.join(output_folder, '1000_270' + filename))

        img_0100.save(os.path.join(output_folder, '0100_' + filename ))
        img_0100_90.save(os.path.join(output_folder, '0100_90' + filename))
        img_0100_180.save(os.path.join(output_folder, '0100_180' + filename))
        img_0100_270.save(os.path.join(output_folder, '0100_270' + filename))

        img_1010.save(os.path.join(output_folder, '1010_' + filename ))
        img_1010_90.save(os.path.join(output_folder, '1010_90' + filename))
        img_1010_180.save(os.path.join(output_folder, '1010_180' + filename))
        img_1010_270.save(os.path.join(output_folder, '1010_270' + filename))
```

We run a for loop to the folder that we stored original images. For images, 11 more new images will be generated. Then, we have to save the new generated images to the folder that we created above. Similarly, the mask data should also perform this step, but the code will not show here since there is only small modification to the code.

Data Preparation for Model Building

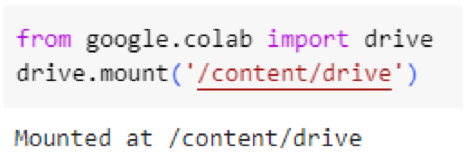
First, we import the python build-in libraries to the notebook.



```python
import os
import glob
import cv2
import numpy as np
from matplotlib import pyplot as plt
import shutil
import pandas as pd

from skimage.io import imread, imshow
from skimage.transform import resize
from skimage.color import rgb2gray

from sklearn.model_selection import train_test_split

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import backend as K
from tensorflow.keras.models import Model, load_model, save_model
from tensorflow.keras.optimizers import Adam, Adamax, RMSprop
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.layers import Input, Activation, BatchNormalization, Dropout, Lambda, Conv2D, Conv2DTranspose, MaxPooling2D, concatenate

# Ignore Warnings
import warnings
warnings.filterwarnings("ignore")
```

The libraries shown is picture above included libraries for data processing like numpy, image processing like skimage, data splitting like train_test_split from sklearn, and model development like keras.

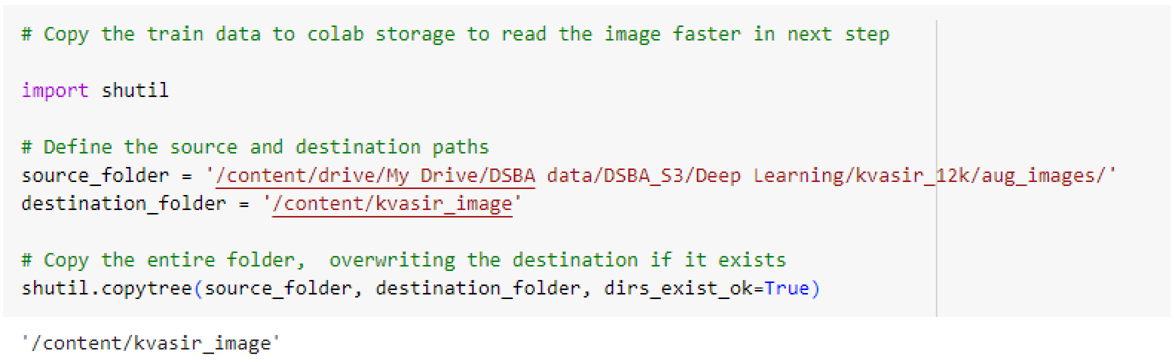
The prepared data was stored in Google Drive, so we have to mount the colab notebook to the Google Drive.



```python
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

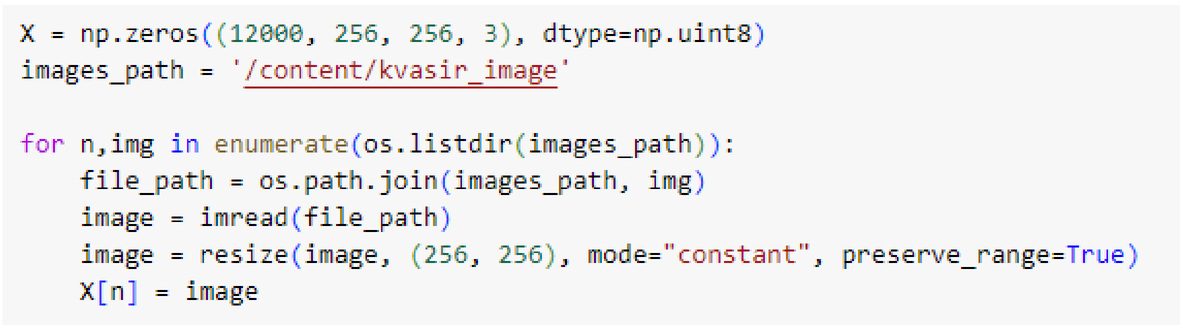
The next step will be importing the image data into this notebook. If we use library 'cv2' to read the image data, this step needs approximately 2 minutes for each image, so we are going to import the images data from Google Drive to the colab storage and read the image data from colab storage.



```python
# Copy the train data to colab storage to read the image faster in next step

import shutil

# Define the source and destination paths
source_folder = '/content/drive/My Drive/DSBA data/DSBA_S3/Deep Learning/kvasir_12k/aug_images/'
destination_folder = '/content/kvasir_image'

# Copy the entire folder, overwriting the destination if it exists
shutil.copytree(source_folder, destination_folder, dirs_exist_ok=True)
```

```
'/content/kvasir_image'
```

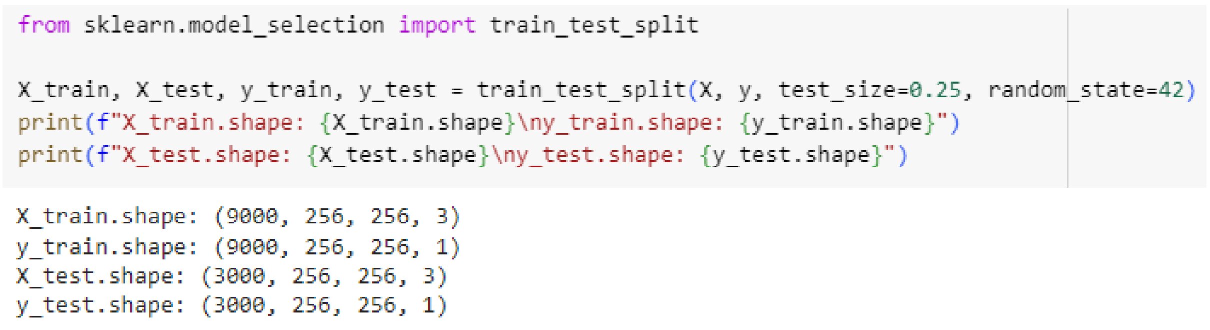
Since the image have size 256x256x3 for each image, we created a numpy array with the image size to store the ‘rgb’ values of the images. The following is the code for training data, the code for mask data is similar, so we are not going to show the code for mask data here.



```
X = np.zeros((12000, 256, 256, 3), dtype=np.uint8)
images_path = '/content/kvasir_image'

for n,img in enumerate(os.listdir(images_path)):
    file_path = os.path.join(images_path, img)
    image = imread(file_path)
    image = resize(image, (256, 256), mode="constant", preserve_range=True)
    X[n] = image
```

Note that for masked data, we converted the channels for each image from ‘rgb’ to grey scale image, so the dimensions for y will be (12000, 256, 256, 1).

Now we have imported the data and ensure that the data type. For this image dataset, we are not going to perform exploratory data analysis (EDA), feature engineering, or data cleaning. So, we can proceed to the train data and test data splitting session.



```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)
print(f"X_train.shape: {X_train.shape}\ny_train.shape: {y_train.shape}")
print(f"X_test.shape: {X_test.shape}\ny_test.shape: {y_test.shape}")
```

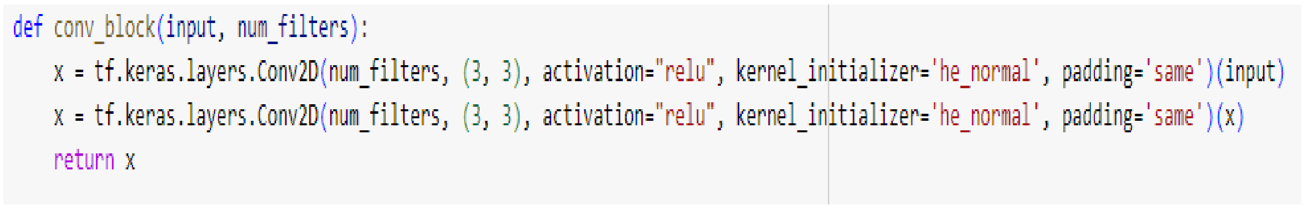
```
X_train.shape: (9000, 256, 256, 3)
y_train.shape: (9000, 256, 256, 1)
X_test.shape: (3000, 256, 256, 3)
y_test.shape: (3000, 256, 256, 1)
```

Clearly, we can see that we have partition the data into train and test with ratio 3:1 with random state 42, this random state will be uses for three different architectures model. From the output cell, we can observe that there are 9000 images prepared for training purpose and 3000 images prepared for validation purpose.

Model Implementation

Original U-net Architecture

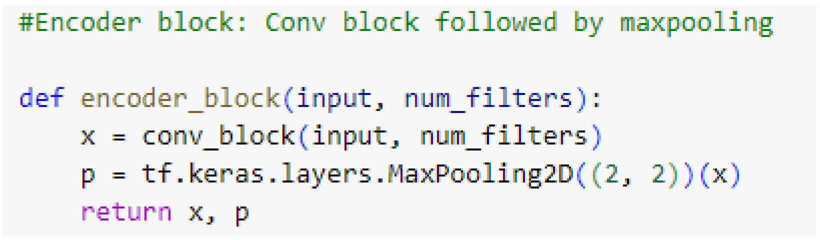
First, we build the original U-net architecture model, then the modified U-net. We first define the necessary functions that apply multiple times in the model, like convolution block, encoder block, and decoder block.



```
def conv_block(input, num_filters):
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(input)
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(x)
    return x
```

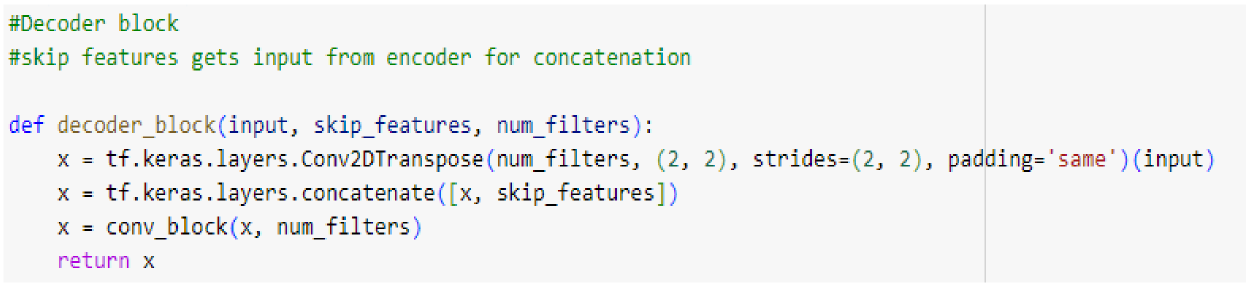
This convolution block function required two inputs which is input image and the number of filters. This function contains 2 steps of convolution operations with kernel size 3x3, activation function will be Rectified Unit, and the padding ‘same’. The output of this function will be the 2-dimensional array after convolution.

The next function to define is the encoder and decoder blocks. First, we define the encoder block. The responsibility of encoder block in this architecture will be proceed mapping to the next layer.



```
#Encoder block: Conv block followed by maxpooling

def encoder_block(input, num_filters):
    x = conv_block(input, num_filters)
    p = tf.keras.layers.MaxPooling2D((2, 2))(x)
    return x, p
```

The first step in this function is applied the function that defined above to proceed convolution operation, then perform a 2-dimensional max pooling with kernel size 2x2. Note that we have two outputs in this function, the output ‘p’ is use for concatenation in the decoder layer. So, the next step will be defining the function for the decoder block, this block mainly responsible for concatenate the output from previous layer and the output ‘p’ from the same encoder layer.



```
#Decoder block
#skip features gets input from encoder for concatenation

def decoder_block(input, skip_features, num_filters):
    x = tf.keras.layers.Conv2DTranspose(num_filters, (2, 2), strides=(2, 2), padding='same')(input)
    x = tf.keras.layers.concatenate([x, skip_features])
    x = conv_block(x, num_filters)
    return x
```

The decoder block includes a convolutional 2-dim transpose operation with kernel size 2x2, strides 2x2, and padding ‘same’, concatenate the two outputs from different layer, then apply convolution function.

To evaluate model performance, we have to define the algorithm for computing intersection over union (IoU). The mathematical formula was shown in methodology section, here we made it in python.

```
# function to create iou coefficient
def iou_coef(y_true, y_pred, smooth=100):
    y_true = tf.cast(y_true, dtype=tf.float32) # Cast y_true to float32
    y_pred = tf.cast(y_pred, dtype=tf.float32) # Cast y_pred to float32
    intersection = K.sum(y_true * y_pred)
    sum = K.sum(y_true + y_pred)
    iou = (intersection + smooth) / (sum - intersection + smooth)
    return iou
```

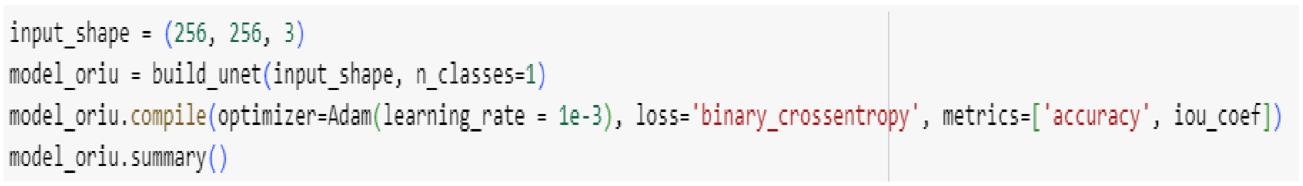
In this function, we have converted the y_true and y_pred into datatype ‘float32’ to match with the output from model. This output of this function is a numerical value between 0 and 1.

Now, we have defined the function that we need. So, we can start to build up the model with the function that we defined above.

```
#Build Unet using the blocks
def build_unet(input_shape, n_classes):
    inputs = Input(input_shape)

    s1, p1 = encoder_block(inputs, 64)
    s2, p2 = encoder_block(p1, 128)
    s3, p3 = encoder_block(p2, 256)
    s4, p4 = encoder_block(p3, 512)

    b1 = conv_block(p4, 1024) #Bridge

    d1 = decoder_block(b1, s4, 512)
    d2 = decoder_block(d1, s3, 256)
    d3 = decoder_block(d2, s2, 128)
    d4 = decoder_block(d3, s1, 64)

    outputs = Conv2D(n_classes, 1, padding="same", activation='sigmoid')(d4)

    model = Model(inputs, outputs, name="U-Net")
    return model
```

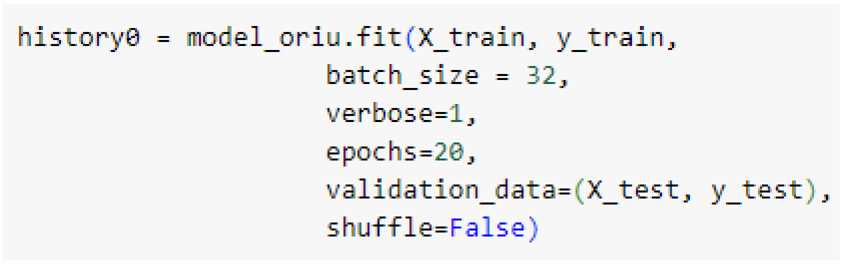
From the code above, we can have built up the original U-net architecture, the ‘outputs’ will generates an image that every pixels are classification of 1 or 0. The activation function before the output will be ‘relu’.

After this, we can set up the hyperparameter for the model before training.



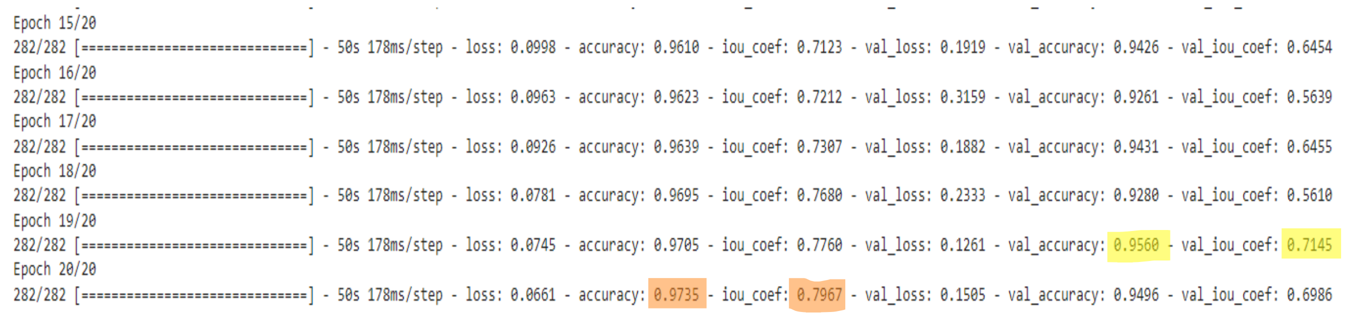
```
input_shape = (256, 256, 3)
model_oriu = build_unet(input_shape, n_classes=1)
model_oriu.compile(optimizer=Adam(learning_rate = 1e-3), loss='binary_crossentropy', metrics=['accuracy', iou_coef])
model_oriu.summary()
```

Here, we have set that learning rate 0.001, loss function is single variable since the target are binary, and the metrics accuracy and IoU. The output of this code will show the architecture and parameter details, since the output was too long and containing too much of information, we leave it in the appendix section. Then, we feed the training image data to the model to training purpose.



```
history0 = model_oriu.fit(X_train, y_train,
                          batch_size = 32,
                          verbose=1,
                          epochs=20,
                          validation_data=(X_test, y_test),
                          shuffle=False)
```

Here we set the batch size 20 to perform mini batch gradient descent under optimizer Adam, and this process will execute 20 epochs.

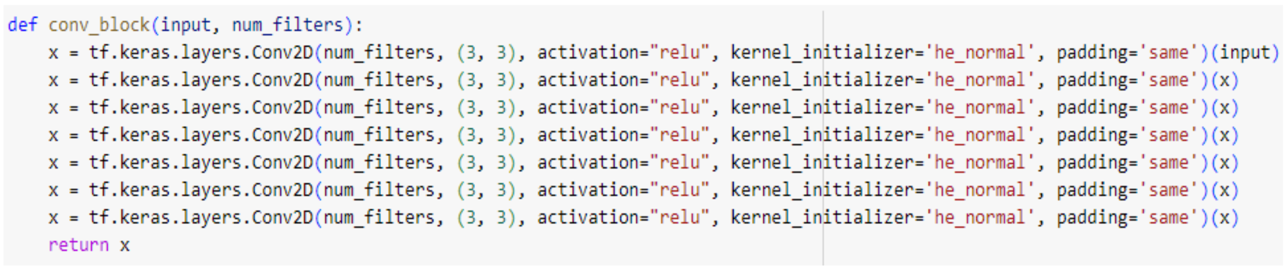


```
Epoch 15/20
282/282 [==============================] - 50s 178ms/step - loss: 0.0998 - accuracy: 0.9610 - iou_coef: 0.7123 - val_loss: 0.1919 - val_accuracy: 0.9426 - val_iou_coef: 0.6454
Epoch 16/20
282/282 [==============================] - 50s 178ms/step - loss: 0.0963 - accuracy: 0.9623 - iou_coef: 0.7212 - val_loss: 0.3159 - val_accuracy: 0.9261 - val_iou_coef: 0.5639
Epoch 17/20
282/282 [==============================] - 50s 178ms/step - loss: 0.0926 - accuracy: 0.9639 - iou_coef: 0.7307 - val_loss: 0.1882 - val_accuracy: 0.9431 - val_iou_coef: 0.6455
Epoch 18/20
282/282 [==============================] - 50s 178ms/step - loss: 0.0781 - accuracy: 0.9695 - iou_coef: 0.7680 - val_loss: 0.2333 - val_accuracy: 0.9280 - val_iou_coef: 0.5610
Epoch 19/20
282/282 [==============================] - 50s 178ms/step - loss: 0.0745 - accuracy: 0.9705 - iou_coef: 0.7760 - val_loss: 0.1261 - val_accuracy: 0.9560 - val_iou_coef: 0.7145
Epoch 20/20
282/282 [==============================] - 50s 178ms/step - loss: 0.0661 - accuracy: 0.9735 - iou_coef: 0.7967 - val_loss: 0.1505 - val_accuracy: 0.9496 - val_iou_coef: 0.6986
```

Above show the model performance based on different metrics, the summarized data will be show in chapter 6 to enhance readability.

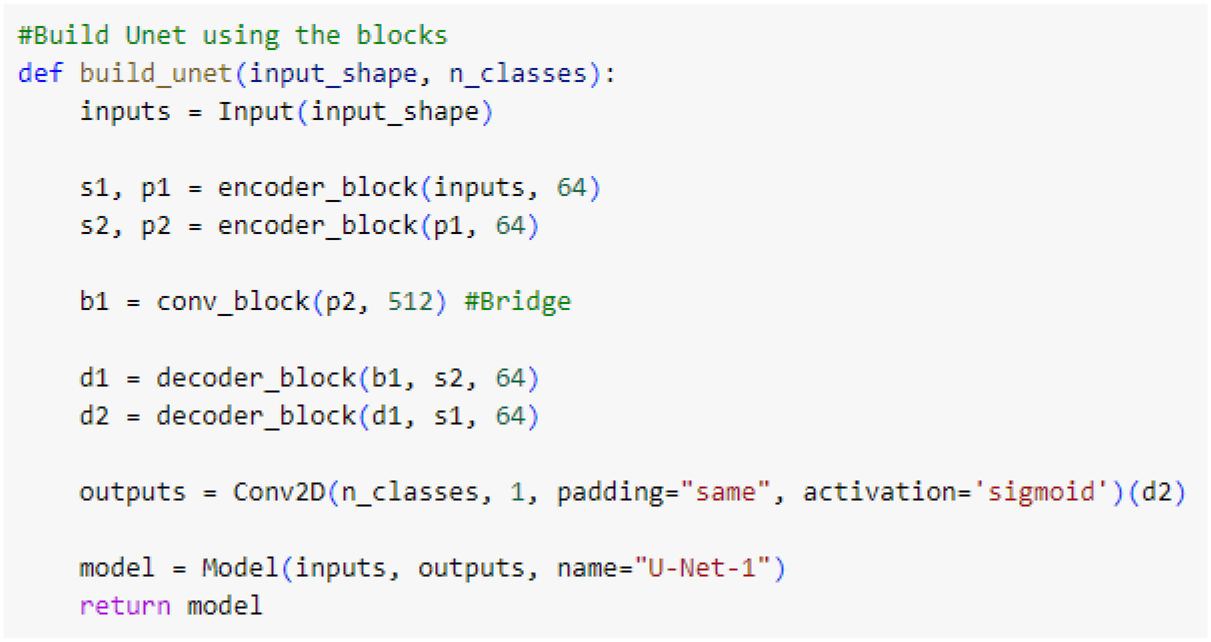
U1 Architecture

In this architecture, we have modified the convolution block, the modified block is shown below.



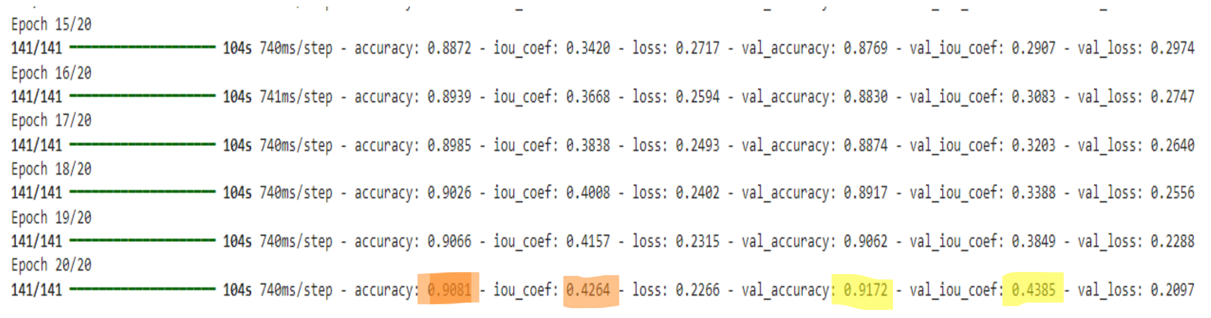
```
def conv_block(input, num_filters):
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(input)
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(x)
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(x)
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(x)
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(x)
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(x)
    x = tf.keras.layers.Conv2D(num_filters, (3, 3), activation="relu", kernel_initializer='he_normal', padding='same')(x)
    return x
```

Clearly, we have added the convolution operation steps on each block. The setting of kernel size, activation function, and padding remains the same with before. Now, we are going to show the model implementation using the latest convolution block and the functions that defined above.



```
#Build Unet using the blocks
def build_unet(input_shape, n_classes):
    inputs = Input(input_shape)

    s1, p1 = encoder_block(inputs, 64)
    s2, p2 = encoder_block(p1, 64)

    b1 = conv_block(p2, 512) #Bridge

    d1 = decoder_block(b1, s2, 64)
    d2 = decoder_block(d1, s1, 64)

    outputs = Conv2D(n_classes, 1, padding="same", activation='sigmoid')(d2)

    model = Model(inputs, outputs, name="U-Net-1")
    return model
```

Since we have decreased the layer in this architecture, the code will be shorter but each layer the steps will be longer and complicated. Since other parts of the code are same with the original U-net model, so we ignored it here and the U2 architecture. Now, we investigate the performance of this architecture.



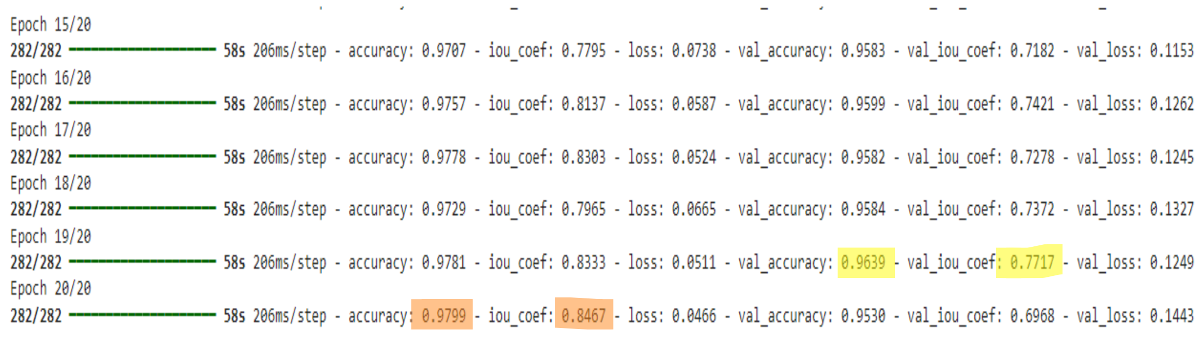
```
Epoch 15/20
141/141 ━━━━━━━━━━━━━━━━━━━━ 104s 740ms/step - accuracy: 0.8872 - iou_coef: 0.3420 - loss: 0.2717 - val_accuracy: 0.8769 - val_iou_coef: 0.2907 - val_loss: 0.2974
Epoch 16/20
141/141 ━━━━━━━━━━━━━━━━━━━━ 104s 741ms/step - accuracy: 0.8939 - iou_coef: 0.3668 - loss: 0.2594 - val_accuracy: 0.8830 - val_iou_coef: 0.3083 - val_loss: 0.2747
Epoch 17/20
141/141 ━━━━━━━━━━━━━━━━━━━━ 104s 740ms/step - accuracy: 0.8985 - iou_coef: 0.3838 - loss: 0.2493 - val_accuracy: 0.8874 - val_iou_coef: 0.3203 - val_loss: 0.2640
Epoch 18/20
141/141 ━━━━━━━━━━━━━━━━━━━━ 104s 740ms/step - accuracy: 0.9026 - iou_coef: 0.4008 - loss: 0.2402 - val_accuracy: 0.8917 - val_iou_coef: 0.3388 - val_loss: 0.2556
Epoch 19/20
141/141 ━━━━━━━━━━━━━━━━━━━━ 104s 740ms/step - accuracy: 0.9066 - iou_coef: 0.4157 - loss: 0.2315 - val_accuracy: 0.9062 - val_iou_coef: 0.3849 - val_loss: 0.2288
Epoch 20/20
141/141 ━━━━━━━━━━━━━━━━━━━━ 104s 740ms/step - accuracy: 0.9081 - iou_coef: 0.4264 - loss: 0.2266 - val_accuracy: 0.9172 - val_iou_coef: 0.4385 - val_loss: 0.2097
```

U2 Architecture

For this architecture, the setup of convolutional block will be same with the convolution block in original U-net architecture, only the model building have different.

```
#Build Unet using the blocks
def build_unet(input_shape, n_classes):
    inputs = Input(input_shape)

    s1, p1 = encoder_block(inputs, 128)
    s2, p2 = encoder_block(p1, 128)
    s3, p3 = encoder_block(p2, 128)
    s4, p4 = encoder_block(p3, 128)
    s5, p5 = encoder_block(p4, 128)

    b1 = conv_block(p5, 1024) #Bridge

    d1 = decoder_block(b1, s5, 128)
    d2 = decoder_block(d1, s4, 128)
    d3 = decoder_block(d2, s3, 128)
    d4 = decoder_block(d3, s2, 128)
    d5 = decoder_block(d4, s1, 128)

    outputs = Conv2D(n_classes, 1, padding="same", activation='sigmoid')(d5)

    model = Model(inputs, outputs, name="U-Net")
    return model
```

This architecture has one more encoder and decoder layer than original U-net architecture, and the number of filters in this architecture was fixed, there are 128 filters in each of the layer excluding bridge. Now, we observe the model performance.



```
Epoch 15/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 206ms/step - accuracy: 0.9707 - iou_coef: 0.7795 - loss: 0.0738 - val_accuracy: 0.9583 - val_iou_coef: 0.7182 - val_loss: 0.1153
Epoch 16/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 206ms/step - accuracy: 0.9757 - iou_coef: 0.8137 - loss: 0.0587 - val_accuracy: 0.9599 - val_iou_coef: 0.7421 - val_loss: 0.1262
Epoch 17/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 206ms/step - accuracy: 0.9778 - iou_coef: 0.8303 - loss: 0.0524 - val_accuracy: 0.9582 - val_iou_coef: 0.7278 - val_loss: 0.1245
Epoch 18/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 206ms/step - accuracy: 0.9729 - iou_coef: 0.7965 - loss: 0.0665 - val_accuracy: 0.9584 - val_iou_coef: 0.7372 - val_loss: 0.1327
Epoch 19/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 206ms/step - accuracy: 0.9781 - iou_coef: 0.8333 - loss: 0.0511 - val_accuracy: 0.9639 - val_iou_coef: 0.7717 - val_loss: 0.1249
Epoch 20/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 206ms/step - accuracy: 0.9799 - iou_coef: 0.8467 - loss: 0.0466 - val_accuracy: 0.9530 - val_iou_coef: 0.6968 - val_loss: 0.1443
```

This model has the best performance across the 3 models no matter in accuracy or IoU metrics. But the number of parameters in this model are the least across these 3 models. Hence, we are going to observe that will this model still has the best performance after hyperparameter tuning process.

Model Tuning and Validation

In this section, we are going to perform hyperparameter tuning for each of the models. The library used in this section will be keras-tuner, so the first step will be installing the keras-tuner library.

```
pip install keras-tuner
```

The hyperparameter that need to tune is different for different architecture of model, each model has different selection of hyperparameter values to tune.

Original U-net Model Tuning

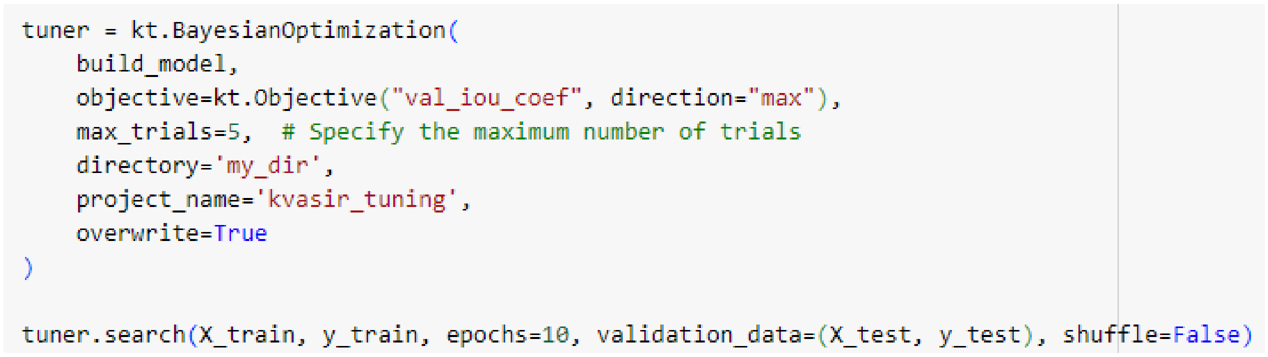
```
import keras_tuner as kt

def build_model(hp):
    input_shape = (256, 256, 3)
    inputs = Input(input_shape)

    # Hyperparameters for the number of filters in each layer
    filters_1 = hp.Int('filters_1', min_value=32, max_value=128, step=16)
    filters_2 = hp.Int('filters_2', min_value=64, max_value=256, step=32)
    filters_3 = hp.Int('filters_3', min_value=128, max_value=512, step=64)
    filters_4 = hp.Int('filters_4', min_value=256, max_value=1024, step=128)

    s1, p1 = encoder_block(inputs, filters_1)
    s2, p2 = encoder_block(p1, filters_2)
    s3, p3 = encoder_block(p2, filters_3)
    s4, p4 = encoder_block(p3, filters_4)

    b1 = conv_block(p4, hp.Int('bridge_filters', min_value=512, max_value=2048, step=128))

    d1 = decoder_block(b1, s4, filters_4)
    d2 = decoder_block(d1, s3, filters_3)
    d3 = decoder_block(d2, s2, filters_2)
    d4 = decoder_block(d3, s1, filters_1)

    outputs = Conv2D(1, 1, padding="same", activation='sigmoid')(d4)

    model = Model(inputs, outputs, name="U-Net")

    # Hyperparameters for learning rate and optimizer
    learning_rate = hp.Choice('learning_rate', values=[1e-2, 3e-3, 8e-4])
    optimizer = hp.Choice('optimizer', values=['adamax', 'rmsprop'])

    if optimizer == 'adamax':
        optimizer = Adamax(learning_rate=learning_rate)
    elif optimizer == 'rmsprop':
        optimizer = RMSprop(learning_rate=learning_rate)

    model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy', iou_coef])
    return model
```

For this original U-net model, we are going to test different number of filters in each layer including bridge, and different choice of learning rate and optimizer. The tables below show the hyperparameters tuning details clearly.

Layer	Min filters	Max filters	Step
1	32	128	16
2	64	256	32
3	128	512	64
4	256	1024	128
bridge	512	2048	128

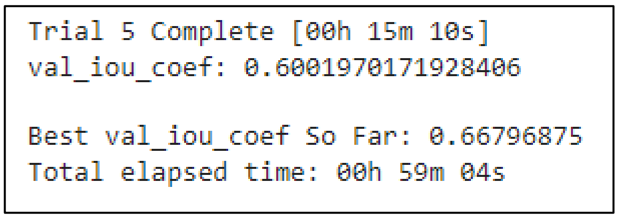
Hyperparameter	Choice 1	Choice 2	Choice 3
Optimizer	Adamax	RMSProp	-
Learning Rate	10^{-2}	3×10^{-3}	8×10^{-4}

Now, we can execute the following code to perform hyperparameter tuning based on the hyperparameters we set above. Since metric ‘accuracy’ is not much important in evaluating image segmentation task, we have assigned the IoU metric to be the objective metric in this process.



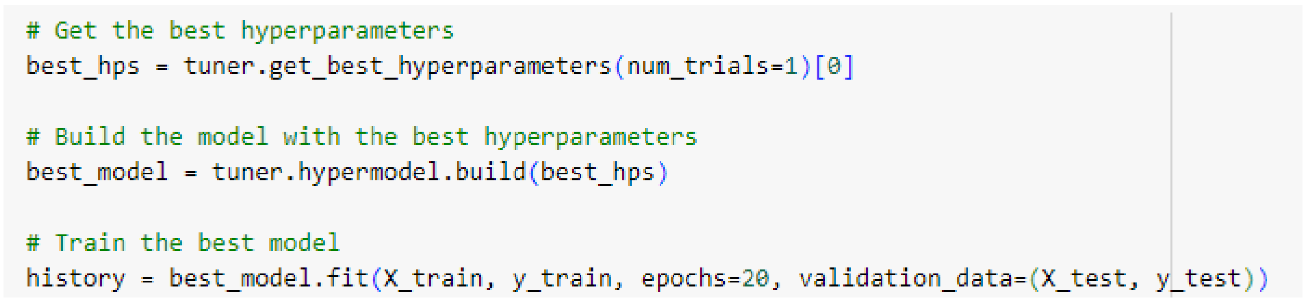
```
tuner = kt.BayesianOptimization(
    build_model,
    objective=kt.Objective("val_iou_coef", direction="max"),
    max_trials=5,  # Specify the maximum number of trials
    directory='my_dir',
    project_name='kvasir_tuning',
    overwrite=True
)

tuner.search(X_train, y_train, epochs=10, validation_data=(X_test, y_test), shuffle=False)
```

Here we have set the maximum trials equal to 5 due to the computer memory restriction, the system will crash if the trial times too much. Here is the output from the code above.



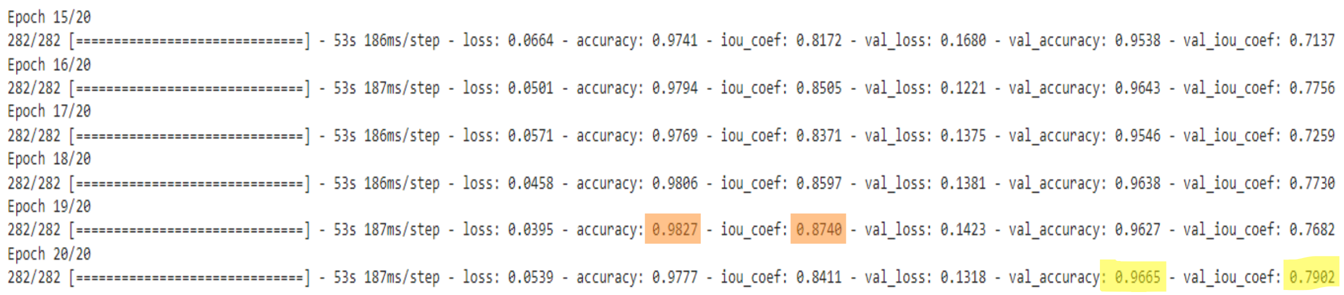
```
Trial 5 Complete [00h 15m 10s]
val_iou_coef: 0.6001970171928406

Best val_iou_coef So Far: 0.66796875
Total elapsed time: 00h 59m 04s
```

Clearly, the best hyperparameter that maximize IoU has been selected. Next, we train a new model using the best hyperparameters chosen from the hyperparameter tuning process.



```
# Get the best hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]

# Build the model with the best hyperparameters
best_model = tuner.hypermodel.build(best_hps)

# Train the best model
history = best_model.fit(X_train, y_train, epochs=20, validation_data=(X_test, y_test))
```

Now, we investigate the result after hyperparameter tuning. Similarly, we only apply 20 epochs since we have to be fair to baseline and tuned model.

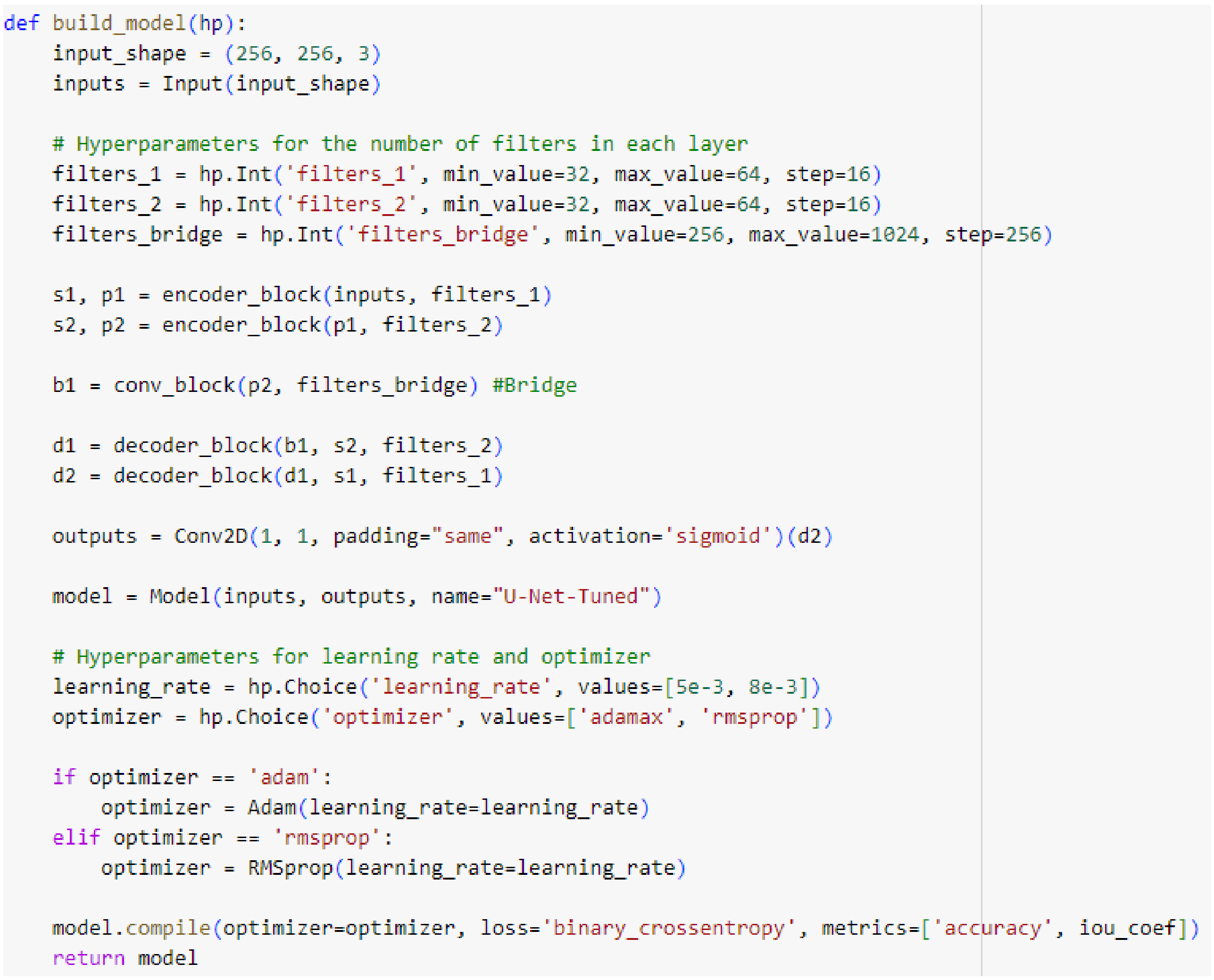


```
Epoch 15/20
282/282 [==============================] - 53s 186ms/step - loss: 0.0664 - accuracy: 0.9741 - iou_coef: 0.8172 - val_loss: 0.1680 - val_accuracy: 0.9538 - val_iou_coef: 0.7137
Epoch 16/20
282/282 [==============================] - 53s 187ms/step - loss: 0.0501 - accuracy: 0.9794 - iou_coef: 0.8505 - val_loss: 0.1221 - val_accuracy: 0.9643 - val_iou_coef: 0.7756
Epoch 17/20
282/282 [==============================] - 53s 186ms/step - loss: 0.0571 - accuracy: 0.9769 - iou_coef: 0.8371 - val_loss: 0.1375 - val_accuracy: 0.9546 - val_iou_coef: 0.7259
Epoch 18/20
282/282 [==============================] - 53s 186ms/step - loss: 0.0458 - accuracy: 0.9806 - iou_coef: 0.8597 - val_loss: 0.1381 - val_accuracy: 0.9638 - val_iou_coef: 0.7730
Epoch 19/20
282/282 [==============================] - 53s 187ms/step - loss: 0.0395 - accuracy: 0.9827 - iou_coef: 0.8740 - val_loss: 0.1423 - val_accuracy: 0.9627 - val_iou_coef: 0.7682
Epoch 20/20
282/282 [==============================] - 53s 187ms/step - loss: 0.0539 - accuracy: 0.9777 - iou_coef: 0.8411 - val_loss: 0.1318 - val_accuracy: 0.9665 - val_iou_coef: 0.7902
```

The IoU score for this model after tuning process achieve 0.79, is significantly increased from the baseline model.

U1 Architecture Model Tuning

The hyperparameter tuning process for U1 and U2 architecture model will used same python library with original U-net model. The code below shows the hyperparameter tuning set-up for U1 model.



```python
def build_model(hp):
    input_shape = (256, 256, 3)
    inputs = Input(input_shape)

    # Hyperparameters for the number of filters in each layer
    filters_1 = hp.Int('filters_1', min_value=32, max_value=64, step=16)
    filters_2 = hp.Int('filters_2', min_value=32, max_value=64, step=16)
    filters_bridge = hp.Int('filters_bridge', min_value=256, max_value=1024, step=256)

    s1, p1 = encoder_block(inputs, filters_1)
    s2, p2 = encoder_block(p1, filters_2)

    b1 = conv_block(p2, filters_bridge) #Bridge

    d1 = decoder_block(b1, s2, filters_2)
    d2 = decoder_block(d1, s1, filters_1)

    outputs = Conv2D(1, 1, padding="same", activation='sigmoid')(d2)

    model = Model(inputs, outputs, name="U-Net-Tuned")

    # Hyperparameters for learning rate and optimizer
    learning_rate = hp.Choice('learning_rate', values=[5e-3, 8e-3])
    optimizer = hp.Choice('optimizer', values=['adamax', 'rmsprop'])

    if optimizer == 'adam':
        optimizer = Adam(learning_rate=learning_rate)
    elif optimizer == 'rmsprop':
        optimizer = RMSprop(learning_rate=learning_rate)

    model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy', iou_coef])
    return model
```

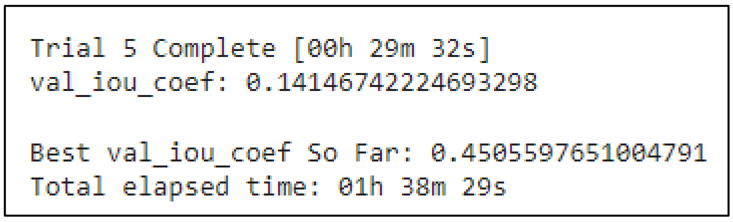
Since U1 model only has 2 layers, we preserve the architecture and edit the number of filters. The setting for change of filters in this architecture are same, which is min 32, max 64, and step distance 16. The learning rate choose for this section will be different with original U-net due to the learning speed in this baseline model. Tables below shows the details.

Layer	Min filters	Max filters	Step
1	32	64	16
2	32	64	16
bridge	256	1024	256

Hyperparameter	Choice 1	Choice 2	Choice 3
Optimizer	Adamax	RMSProp	-
Learning Rate	5×10^{-3}	8×10^{-3}	-

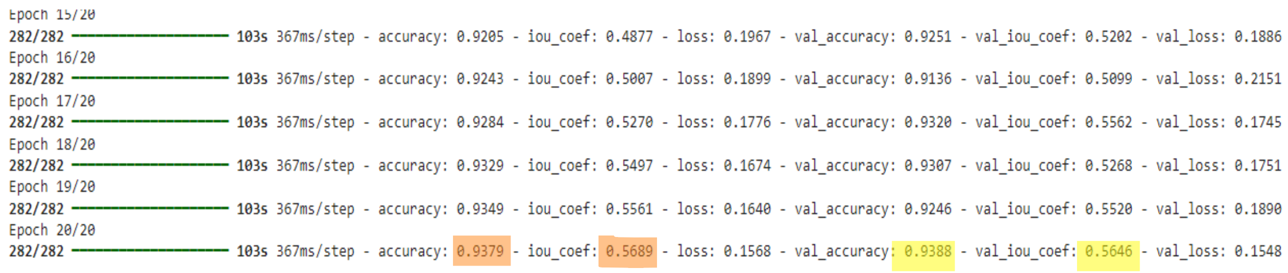
-

Using the same code from above to perform tuner search, we will get the result as follow.



```
Trial 5 Complete [00h 29m 32s]
val_iou_coef: 0.14146742224693298

Best val_iou_coef So Far: 0.4505597651004791
Total elapsed time: 01h 38m 29s
```

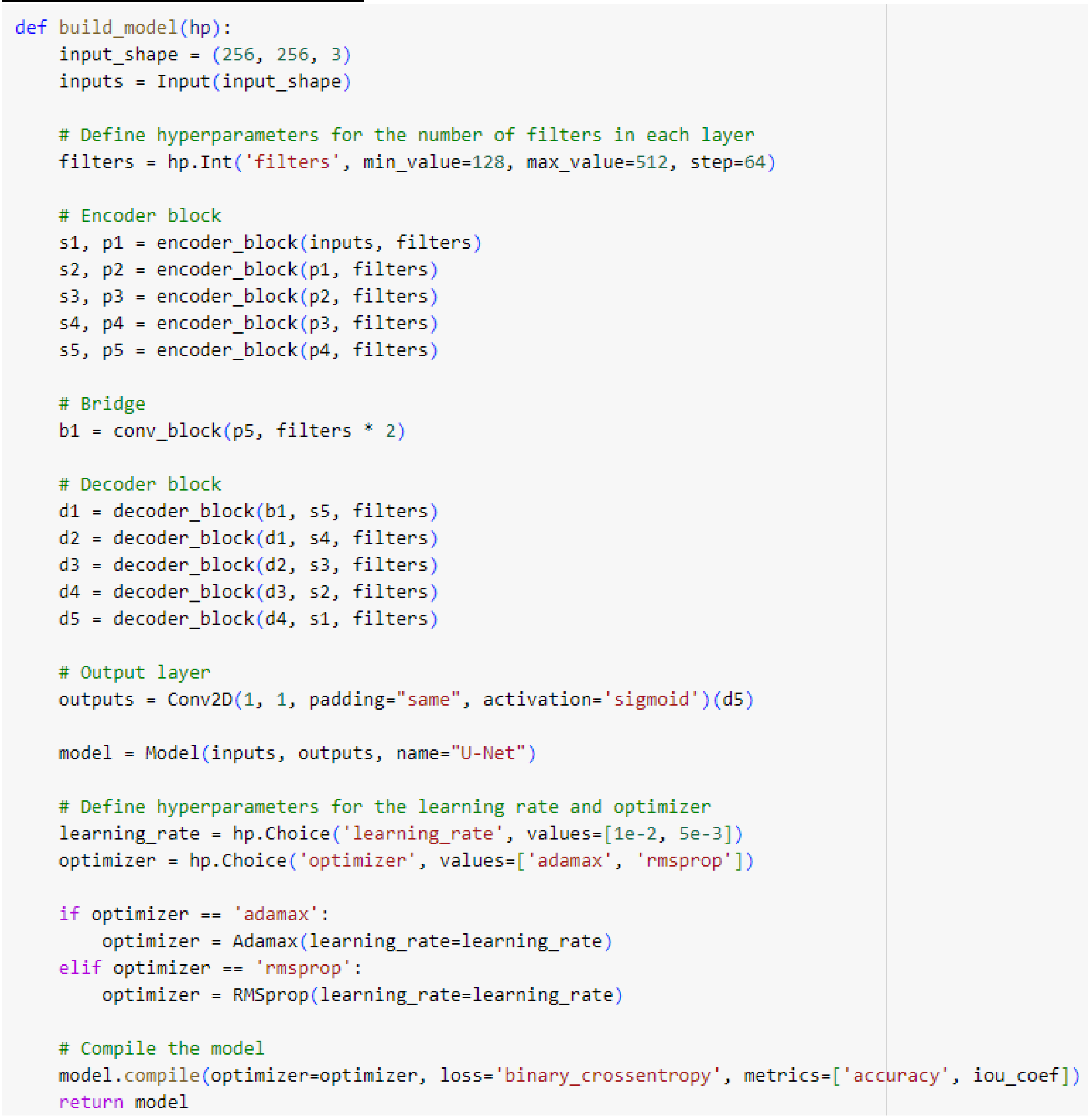
Now, we train the model with best hyperparameters.



```
Epoch 15/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 103s 367ms/step - accuracy: 0.9205 - iou_coef: 0.4877 - loss: 0.1967 - val_accuracy: 0.9251 - val_iou_coef: 0.5202 - val_loss: 0.1886
Epoch 16/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 103s 367ms/step - accuracy: 0.9243 - iou_coef: 0.5007 - loss: 0.1899 - val_accuracy: 0.9136 - val_iou_coef: 0.5099 - val_loss: 0.2151
Epoch 17/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 103s 367ms/step - accuracy: 0.9284 - iou_coef: 0.5270 - loss: 0.1776 - val_accuracy: 0.9320 - val_iou_coef: 0.5562 - val_loss: 0.1745
Epoch 18/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 103s 367ms/step - accuracy: 0.9329 - iou_coef: 0.5497 - loss: 0.1674 - val_accuracy: 0.9307 - val_iou_coef: 0.5268 - val_loss: 0.1751
Epoch 19/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 103s 367ms/step - accuracy: 0.9349 - iou_coef: 0.5561 - loss: 0.1640 - val_accuracy: 0.9246 - val_iou_coef: 0.5520 - val_loss: 0.1890
Epoch 20/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 103s 367ms/step - accuracy: 0.9379 - iou_coef: 0.5689 - loss: 0.1568 - val_accuracy: 0.9388 - val_iou_coef: 0.5646 - val_loss: 0.1548
```

The IoU score increase from 0.4385 to 0.5646 by performing hyperparameter tuning, but still not good enough by comparing to other models.

U2 Architecture Model Tuning



```python
def build_model(hp):
    input_shape = (256, 256, 3)
    inputs = Input(input_shape)

    # Define hyperparameters for the number of filters in each layer
    filters = hp.Int('filters', min_value=128, max_value=512, step=64)

    # Encoder block
    s1, p1 = encoder_block(inputs, filters)
    s2, p2 = encoder_block(p1, filters)
    s3, p3 = encoder_block(p2, filters)
    s4, p4 = encoder_block(p3, filters)
    s5, p5 = encoder_block(p4, filters)

    # Bridge
    b1 = conv_block(p5, filters * 2)

    # Decoder block
    d1 = decoder_block(b1, s5, filters)
    d2 = decoder_block(d1, s4, filters)
    d3 = decoder_block(d2, s3, filters)
    d4 = decoder_block(d3, s2, filters)
    d5 = decoder_block(d4, s1, filters)

    # Output layer
    outputs = Conv2D(1, 1, padding="same", activation='sigmoid')(d5)

    model = Model(inputs, outputs, name="U-Net")

    # Define hyperparameters for the learning rate and optimizer
    learning_rate = hp.Choice('learning_rate', values=[1e-2, 5e-3])
    optimizer = hp.Choice('optimizer', values=['adamax', 'rmsprop'])

    if optimizer == 'adamax':
        optimizer = Adamax(learning_rate=learning_rate)
    elif optimizer == 'rmsprop':
        optimizer = RMSprop(learning_rate=learning_rate)

    # Compile the model
    model.compile(optimizer=optimizer, loss='binary_crossentropy', metrics=['accuracy', iou_coef])
    return model
```

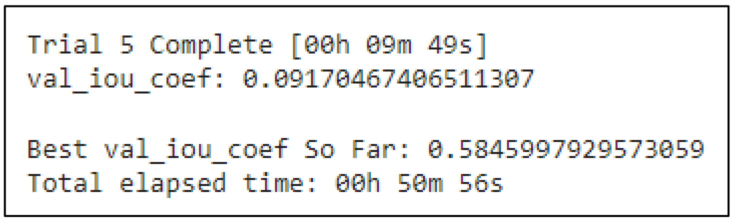
The code for performing hyperparameter tuning for this architecture will be longer due to the design of this architecture. The number of filters is same for each layer exclude the bridge. The learning rate choose for this section will be different with original U-net due to the learning speed in this baseline model. Tables below shows the details.

Layer	Min filters	Max filters	Step
1	128	512	64
2			
3			
4			
5			
bridge	256	1024	128

Note that in this case, the filters of bridge will be decided by the number of filters in the encoder/decoder layer. More precisely, the filters for bridge will be the number of filters at decoder layers multiple by 2.

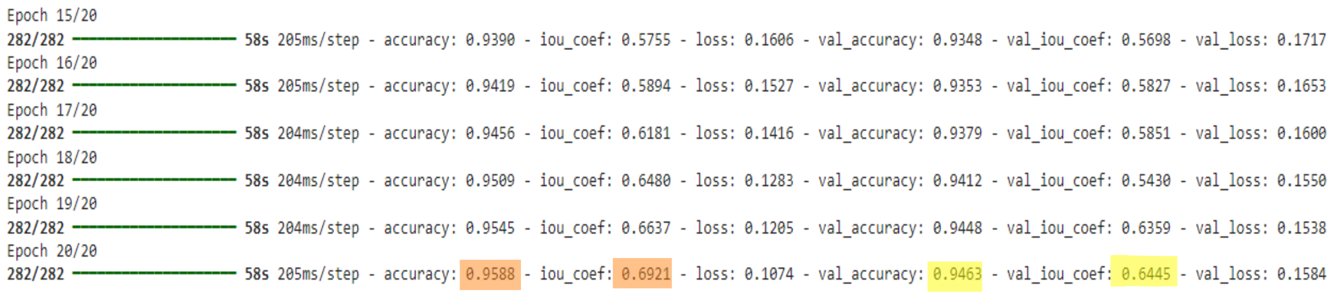
Hyperparameter	Choice 1	Choice 2	Choice 3
Optimizer	Adamax	RMSProp	-
Learning Rate	10^{-2}	5×10^{-3}	-

Using the same code from above to perform tuner search, we will get the result as follow.



```
Trial 5 Complete [00h 09m 49s]
val_iou_coef: 0.09170467406511307

Best val_iou_coef So Far: 0.5845997929573059
Total elapsed time: 00h 50m 56s
```

Now, we train the model with best hyperparameters.

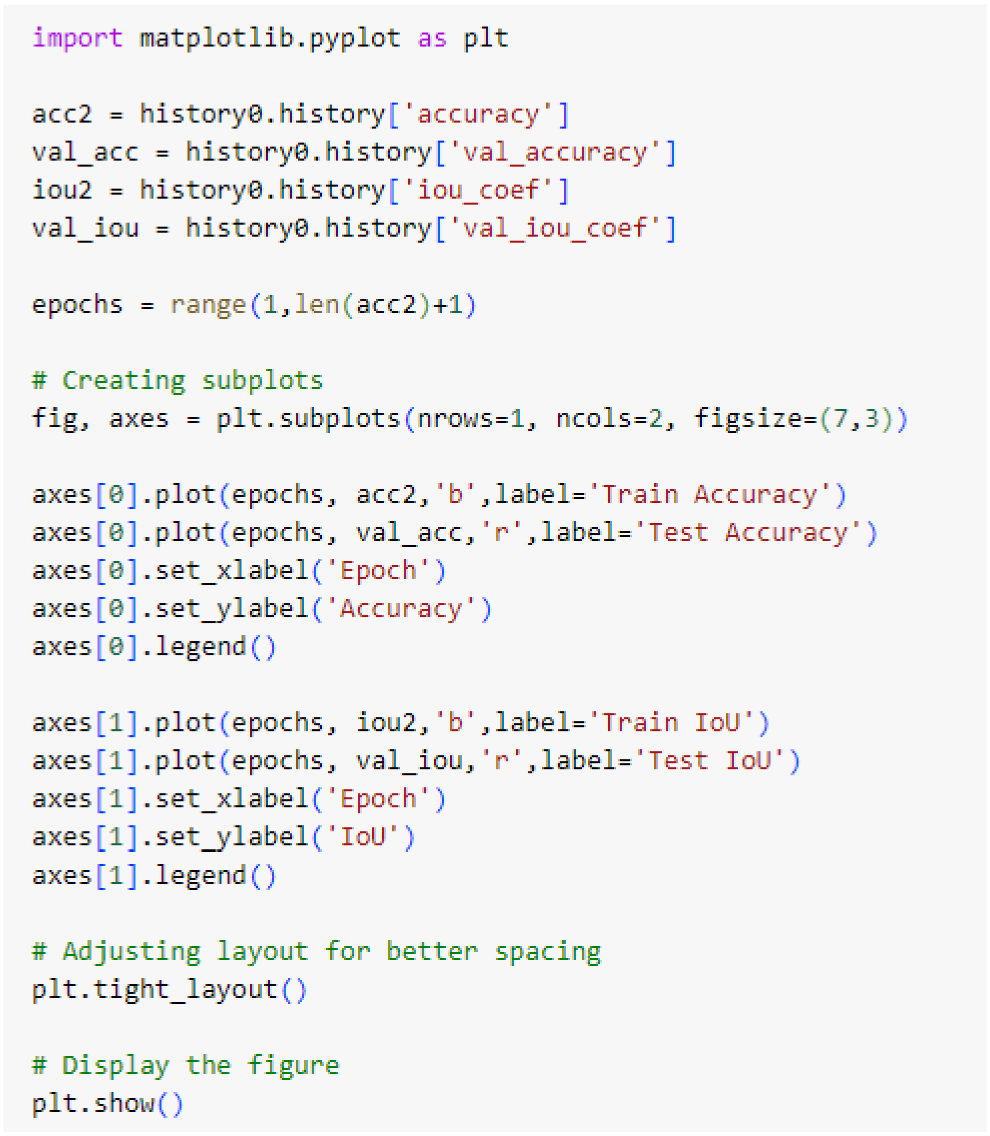


```
Epoch 15/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 205ms/step - accuracy: 0.9390 - iou_coef: 0.5755 - loss: 0.1606 - val_accuracy: 0.9348 - val_iou_coef: 0.5698 - val_loss: 0.1717
Epoch 16/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 205ms/step - accuracy: 0.9419 - iou_coef: 0.5894 - loss: 0.1527 - val_accuracy: 0.9353 - val_iou_coef: 0.5827 - val_loss: 0.1653
Epoch 17/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 204ms/step - accuracy: 0.9456 - iou_coef: 0.6181 - loss: 0.1416 - val_accuracy: 0.9379 - val_iou_coef: 0.5851 - val_loss: 0.1600
Epoch 18/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 204ms/step - accuracy: 0.9509 - iou_coef: 0.6480 - loss: 0.1283 - val_accuracy: 0.9412 - val_iou_coef: 0.5430 - val_loss: 0.1550
Epoch 19/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 204ms/step - accuracy: 0.9545 - iou_coef: 0.6637 - loss: 0.1205 - val_accuracy: 0.9448 - val_iou_coef: 0.6359 - val_loss: 0.1538
Epoch 20/20
282/282 ━━━━━━━━━━━━━━━━━━━━ 58s 205ms/step - accuracy: 0.9588 - iou_coef: 0.6921 - loss: 0.1074 - val_accuracy: 0.9463 - val_iou_coef: 0.6445 - val_loss: 0.1584
```

The baseline model for U2 architecture achieved 0.77 of IoU score but the IoU score after tuning is only 0.6445. The probability of this happen is quite small but still possible, means that the hyperparameter we have chosen when building baseline model is good enough.

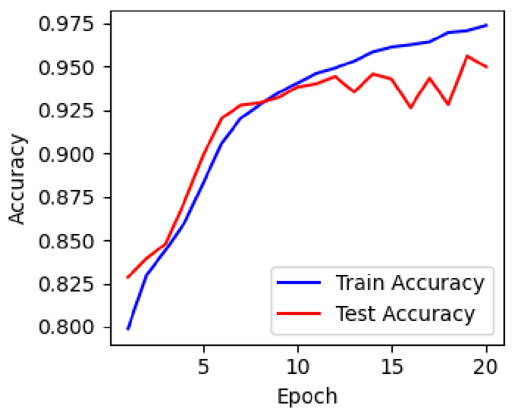
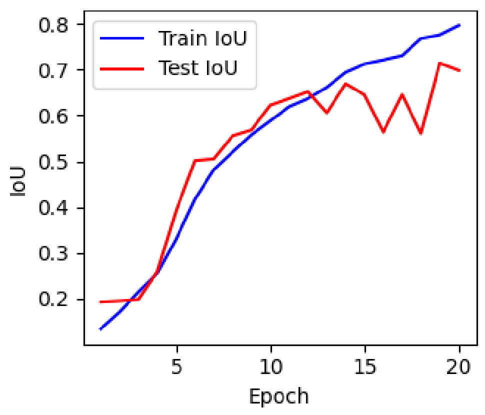
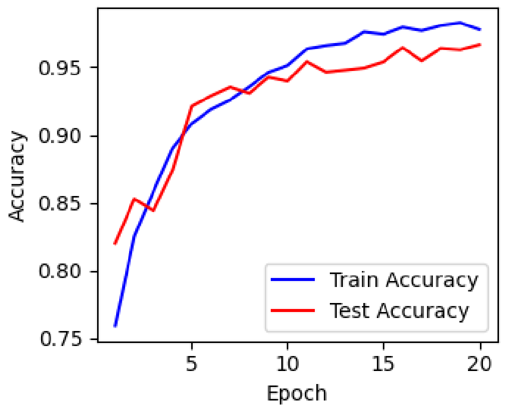
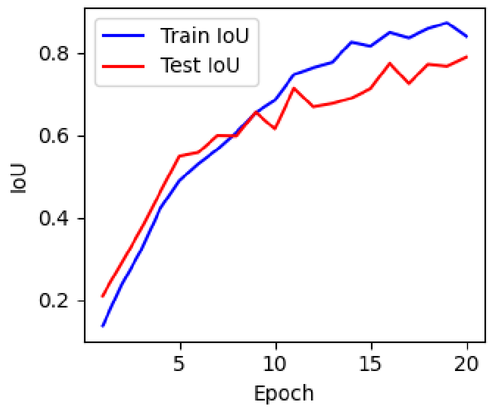
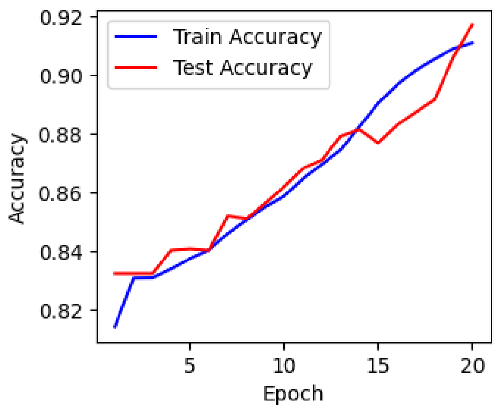
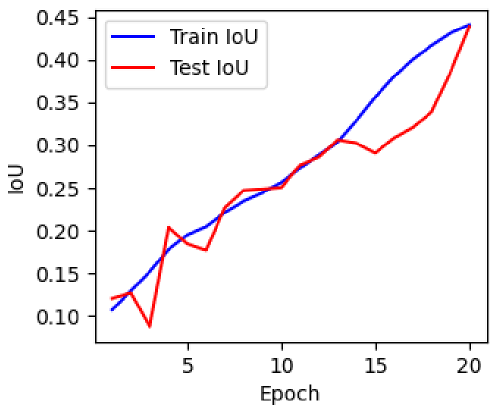
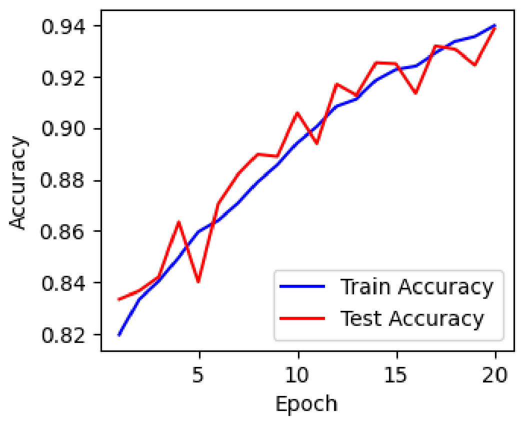
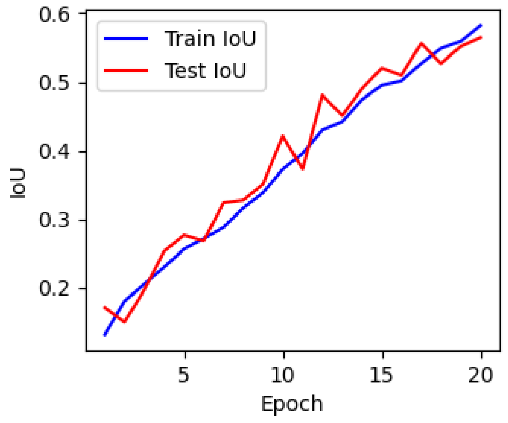
Visualization and Critical Analysis

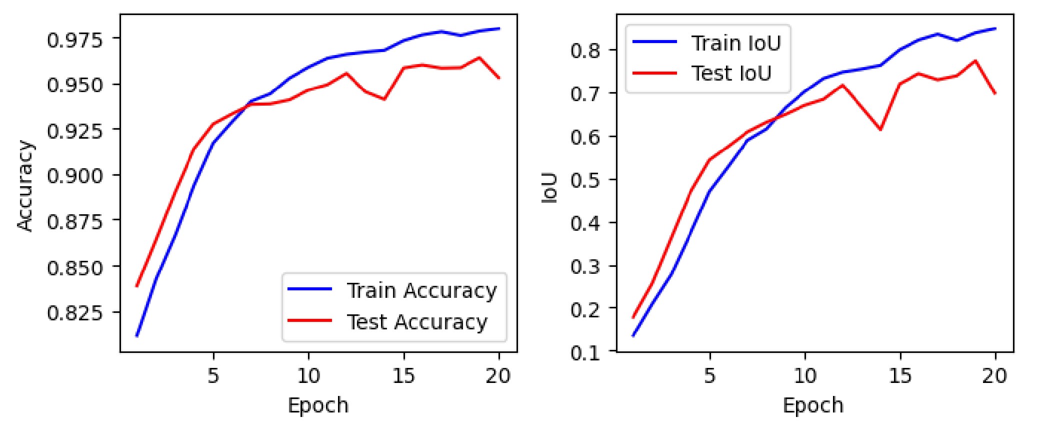
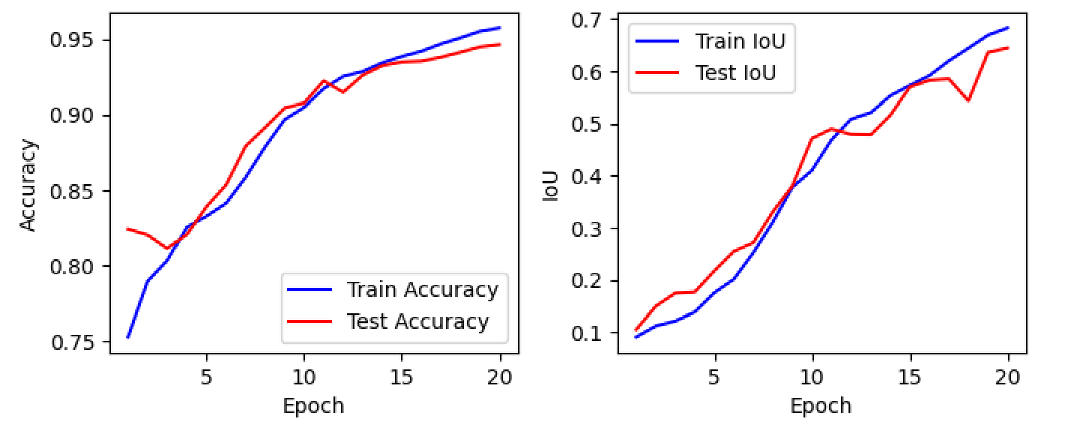
In the previous section, we have implemented 3 different models and its hyperparameter tuning. The accuracy and IoU metric were defined for model performance evaluations. The code below shows how to plot the trends between accuracy and IoU score for training and testing dataset.



```
import matplotlib.pyplot as plt

acc2 = history0.history['accuracy']
val_acc = history0.history['val_accuracy']
iou2 = history0.history['iou_coef']
val_iou = history0.history['val_iou_coef']

epochs = range(1,len(acc2)+1)

# Creating subplots
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(7,3))

axes[0].plot(epochs, acc2,'b',label='Train Accuracy')
axes[0].plot(epochs, val_acc,'r',label='Test Accuracy')
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Accuracy')
axes[0].legend()

axes[1].plot(epochs, iou2,'b',label='Train IoU')
axes[1].plot(epochs, val_iou,'r',label='Test IoU')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('IoU')
axes[1].legend()

# Adjusting layout for better spacing
plt.tight_layout()

# Display the figure
plt.show()
```

Since we have implemented few models, WLOG, the code above showing the plot for original U-net baseline model, code for others architecture is similar. The result from the output will show in the next page.

Model	Accuracy and IoU	
Ori U-net		
Tuned Ori U-net		
U1		
Tuned U1		

U2	
Tuned U2	

Clearly, we can see that the blue colour line (train) is higher than red line (test) at most of the time, this shows the training is consistency. The table below show the model performance of each model precisely. Note that all the models are fit until epochs 20.

Model	Train Accuracy	Train IoU	Train Loss	Test Accuracy	Test IoU	Test Loss
Ori U-net	.9735	.7967	.0661	.9496	.6986	.1505
Tuned Ori	.9777	.8411	.0539	**.9665**	**.7902**	**.1318**
U1	.9081	.4264	.2266	.9172	.4385	.2097
Tuned U1	.9379	.5689	.1568	.9388	.5646	.1548
U2	**.9799**	**.8467**	**.0466**	.9530	.6968	.1443
Tuned U2	.9588	.6921	.1074	.9463	.6445	.1584

From the table above, we found that U2 architecture has the best performance at training set, the training accuracy, Iou, and loss is the best across all the models. But, tuned original U-net has the best performance at testing set in all the evaluation metrics. U2 architecture has very good performance during training set, for the hyperparameter tuning, we didn’t get better performance compared to the baseline model, this is because the hyperparameter selection during training is good enough, and we didn’t find a better hyperparameter for this dataset. For the original U-net model, the baseline model performance is not good, but the performance boosted after performing hyperparameter tuning. Overall, U1 architecture has the worst performance across these 3 different architectures, no matter baseline model or after hyperparameter tuning. This shows that decrease the encoder/decoder layer and increase the number of convolutions will not improve the model performance. Even the result of tuned U2 architecture is not better than tuned original U-net architecture, but the result was close. This shows that increase the number of encoder/decoders helps to improve the model performance.

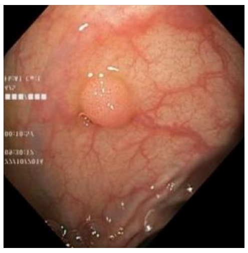
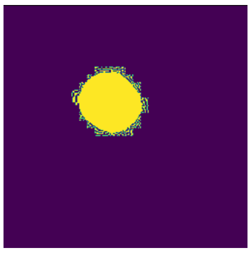
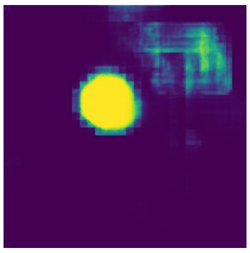
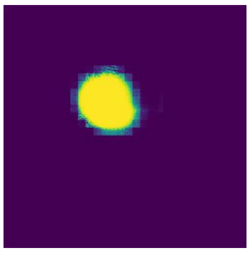
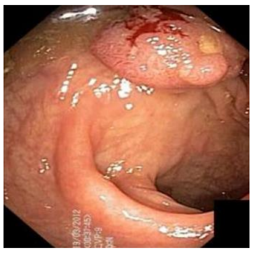
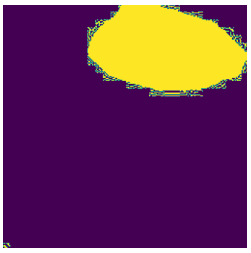
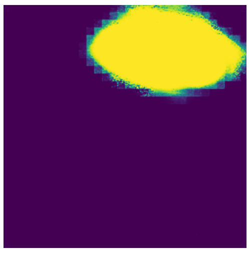
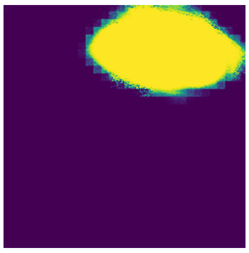
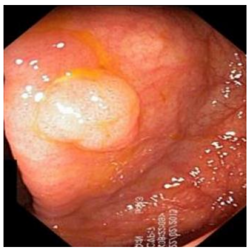
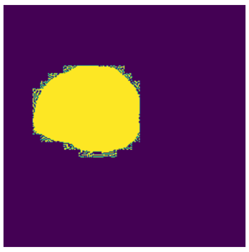
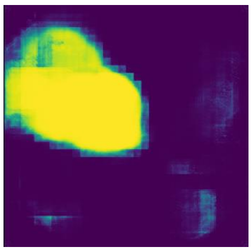
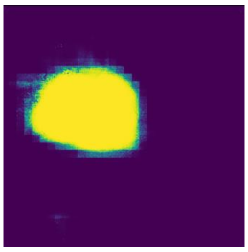
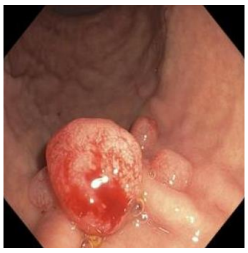
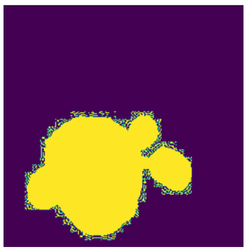
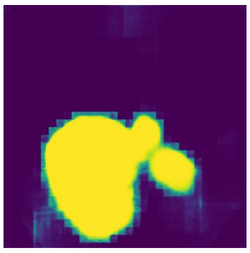
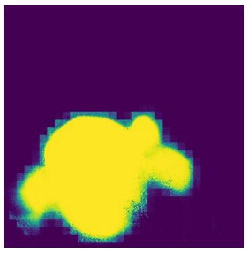
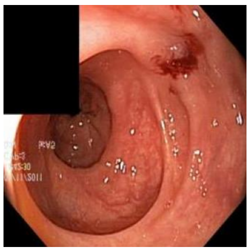
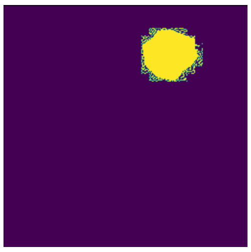
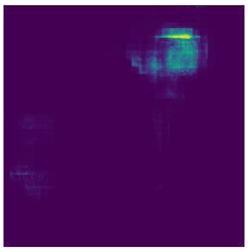
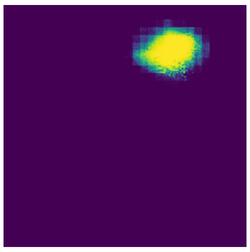
When we discuss the model performance, the complexity of the model is one of the important topics to discuss. From the model results, we can easily to observe that original U-net model has the best performance in testing dataset. However, the original U-net model has the highest number of trainable parameters, this means that this architecture is the most complexity model in this experiment. The table below shows the number of trainable parameters across different architectures.

Model	Trainable Parameters	Memory
Ori U-net	31,031,745	118.38
U1	15,674,497	59.79
U2	14,951,041	57.03

From the table above, we can observe that original U-net architecture has the most trainable parameters, but the model performance is only slightly better than the U2 architecture model. From business approaches, U2 architecture will be more suitable for applications since the performance is guaranteed and the computational cost to train the model is not high as original U-net model.

<u>Visualization of Model Prediction</u>

The following table shows some examples output from the model. Since original U-net model has the best performance in this experiment, and the difference between baseline and tuned models is obviously, so we observe the output from this model.

Image	Truth Mask	Mask from Baseline Model	Mask from Tuned Model
			
			
			
			
			

From the example above, the output from baseline model is still need improvement, the output of tuned model is very close to the truth mask image. In medical domain, the mask can be very important because we are not only curious about the position of tumor, we also need to know the size of the tumor.