RGB-D DEPTH ESTIMATION

U-Net segmentation

Authors:

Author-1: Abdullah Abid 362376

Author-2: Khubaib Hidar Khan 363126

Author-3: Muhammad Faizan Faiz 362940

SMME-NUST

RIME-833 DEEP LEARNING 2022

Enhancing Depth Estimation from RGB-D Images with U-Net Semantic Segmentation and Model Ensembling

Abstract:

Depth estimation from RGB-D images is a crucial task in computer vision, with applications in robotics, augmented reality, and more. In this paper, we propose a method for improved depth estimation using U-Net semantic segmentation and model Ensembling. Our method involves using U-Net to segment the RGB images into semantically meaningful Depth images, and then using the segmentation masks to refine the depth estimates. We also employ model ensembling to combine the predictions of 3 depth estimation models, further improving the accuracy of our approach. We evaluate our method on a variety of RGB-D datasets and demonstrate significant improvements in accuracy over the state-of-the-art. Our approach is fast, robust, and can be easily integrated into a variety of depth estimation pipelines.

Introduction:

With a wide range of applications of Depth Estimation including robotics, augmented reality, and 3D modeling. Accurate depth information allows for a better understanding of the spatial layout and structure of a scene, enabling tasks such as object recognition, scene reconstruction, and navigation.

In this report, we present an approach for improving depth estimation from RGB-D images using U-Net semantic segmentation and model ensembling. We evaluate our approach on the “NYU Depth V2” dataset, a large and diverse dataset of indoor scenes captured with a Kinect camera. The dataset consists of around 50 thousands RGB images, each annotated with a dense depth image map in a grayscale ”the closest the object to the camera the darker the pixel ” .

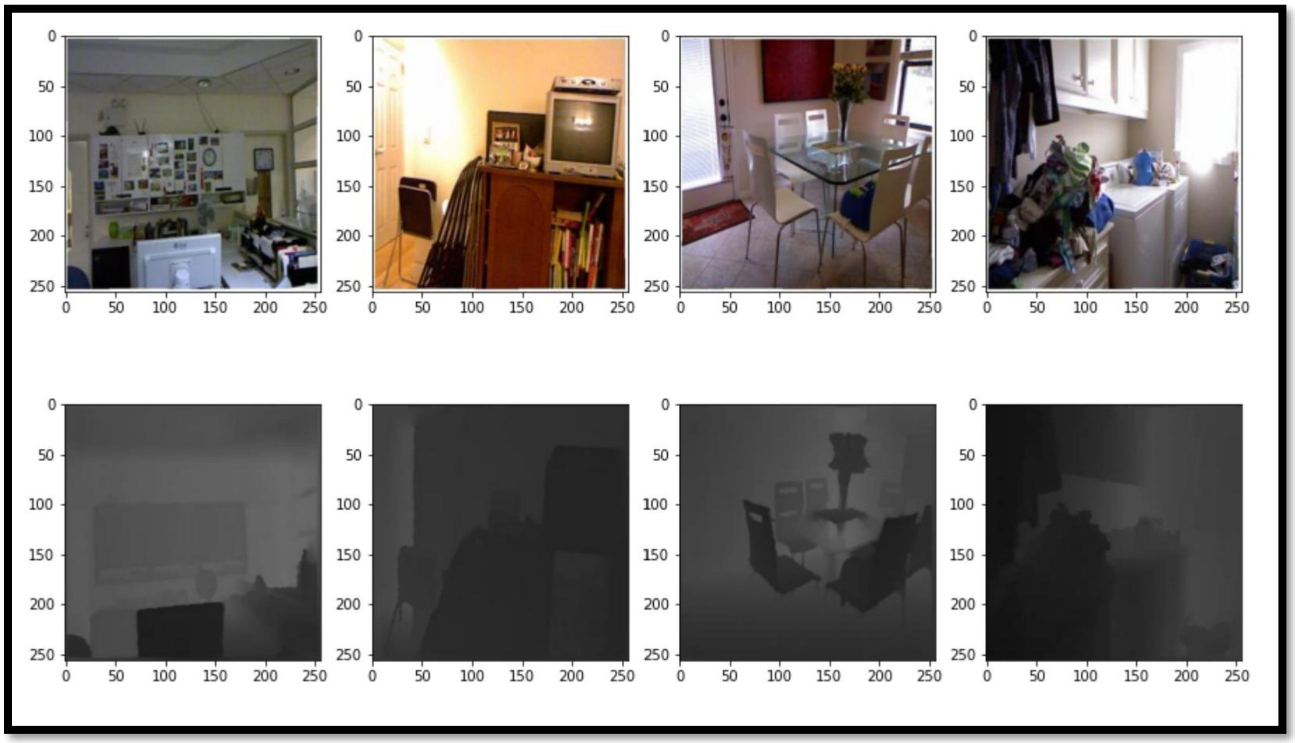


Figure 1.1 : Samples of Images and Masks from the data

Our approach involves using U-Net to segment the RGB-D images into semantically meaningful regions, and then using the segmentation masks to refine the depth estimates. We also employ model ensembling to combine the predictions of multiple depth estimation models, further improving the accuracy of our approach. We compare our method to several state-of-the-art depth estimation methods and demonstrate significant improvements in accuracy.

In the following sections, we explain briefly U-net and model ensembling.

U-net segmentation:

U-Net is a deep learning architecture developed for image segmentation tasks. It was originally developed for biomedical image segmentation but has since been applied to a wide range of image segmentation tasks in computer vision.

U-Net architecture consist of:

- Encoder: which is nothing but a CNN that down sample the images into features
- Concatenation layer: this layer is used to assemble all the features extracted from the encoder into one tensor.
- Skip connections: those are some tokens assigned at the Encoder level so it would be fed to the Decoder again to retain the size and pixels positions.
- Decoder: this Decoder Up-samples the features from the previous step using transposed convolution, and finally construct an image out of those features.

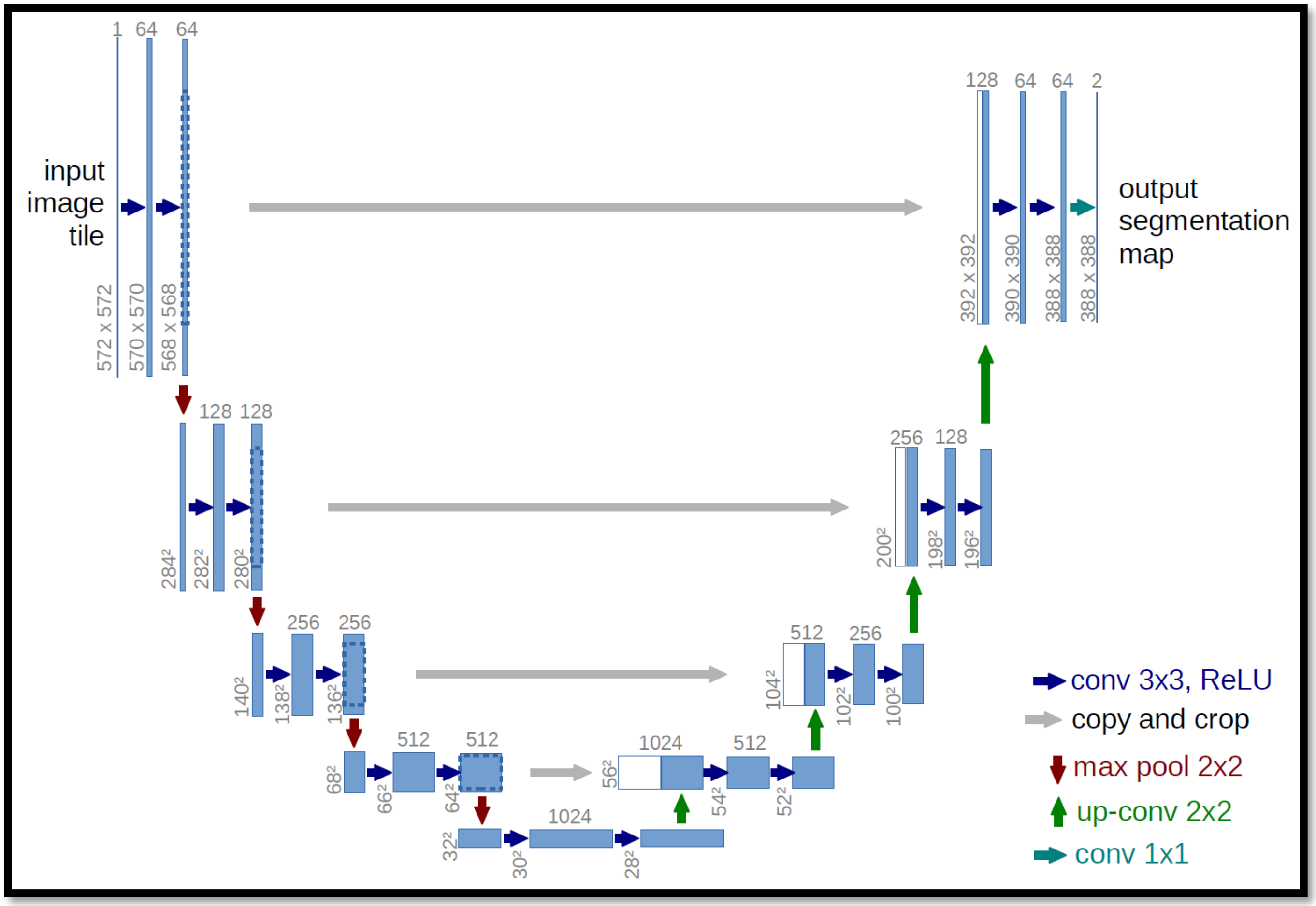


Figure 1.2 : U-net architecture from the original paper

Model ensemble technique:

Model ensembling is a machine learning technique that combines the predictions of multiple models to achieve improved performance. The basic idea is that by combining the predictions of several models, we can reduce the variance and improve the generalizability of the final prediction.

There are several ways to ensemble models, including:

- Voting: In a voting ensemble, each model makes a prediction, and the final prediction is the class or label that received the most votes. This is a simple and effective way to ensemble models, but it is sensitive to the diversity of the models.
- Weighted voting: In a weighted voting ensemble, each model is assigned a weight based on its performance, and the final prediction is based on the weighted votes of the models. This allows the ensemble to assign more importance to models that are more accurate.
- Averaging: In an averaging ensemble, the predictions of the models are averaged to produce the final prediction. This works well for continuous targets, such as regression problems.
- Stacking: In a stacking ensemble, a meta-model is trained to make a final prediction based on the predictions of the base models. The base models can be of different types, and the meta-model can be trained using techniques such as linear regression or a neural network.

Model ensembling is a powerful technique that has been used to achieve state-of-the-art results in many machine learning competitions. It is especially useful when the base models are diverse and complementary, as it allows them to "vote" on the final prediction, increasing the accuracy and robustness of the ensemble.

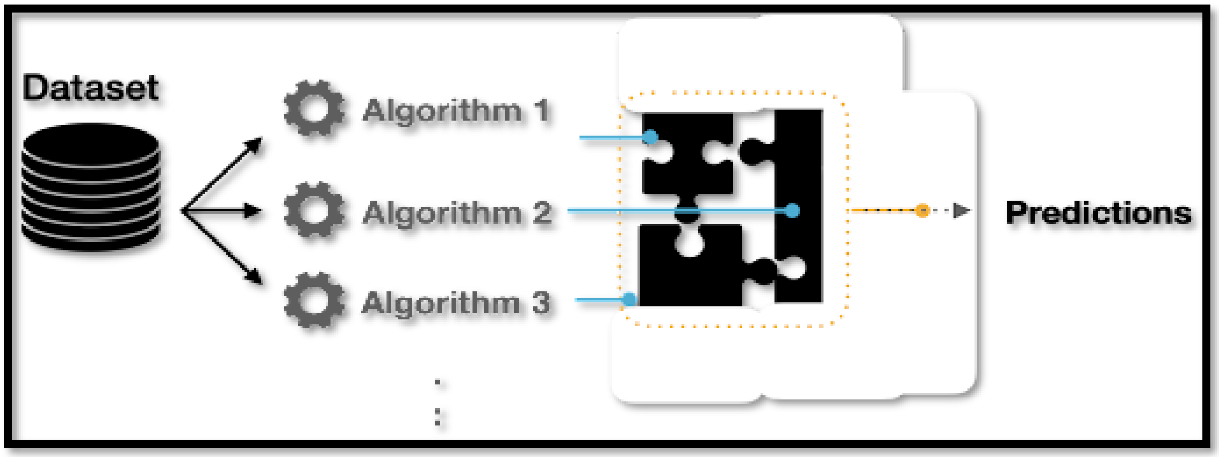


Figure 1.3: Ensembling Block diagram

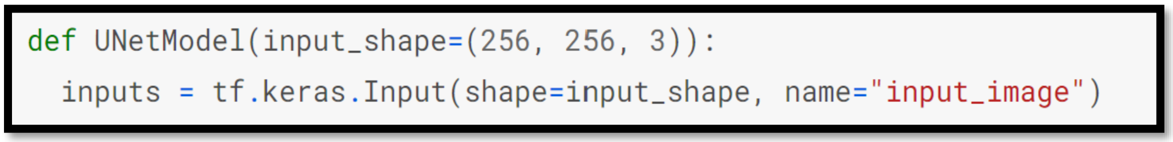
Model Discussion:

The following is a detailed discussion about model architecture, we undertake the following:

1. Input tensor.
2. Encoder.
3. Decoder.
4. Model.
5. Loss functions Utilized.
6. Optimization technique.
7. Results of each model.
8. Model ensembling.
9. Final Results.

1-Input Tensor:

We used the same input tensor for all of the 3 models we developed, generally the shape of the tensor:



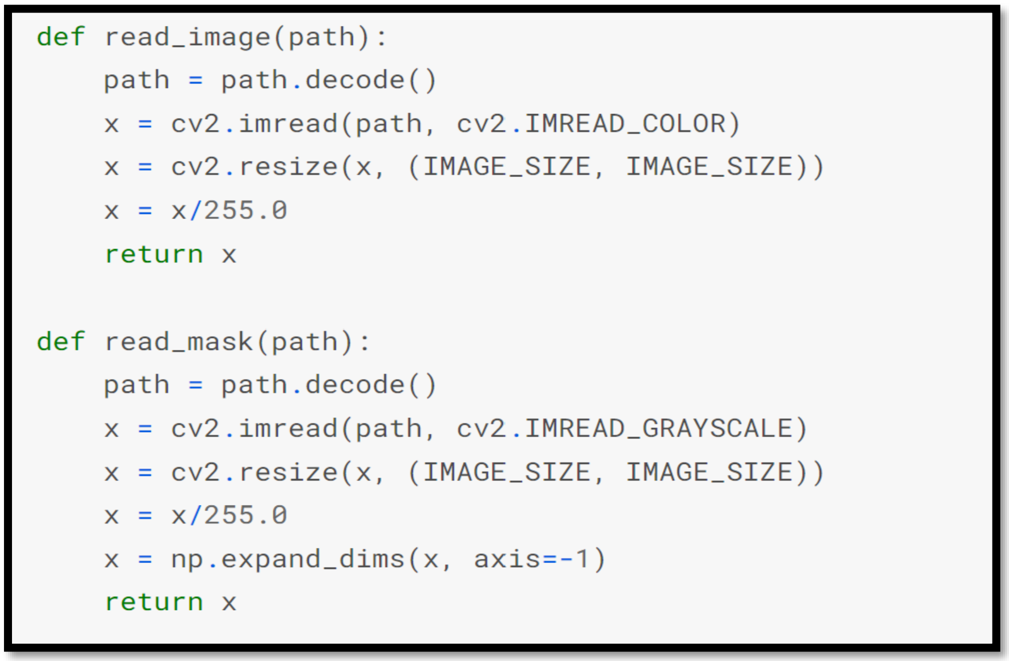
```python
def UNetModel(input_shape=(256, 256, 3)):
  inputs = tf.keras.Input(shape=input_shape, name="input_image")
```

Code snippet 1.1: Input shape

[(Image Hight x image Width x channels) x Batch_size]

[(256 x 256 x 3) x 16]

Given that the images and masks has already been pre-processed and resized into (256 X 256) and the values for each channel is normalized by dividing each pixel by 255



```python
def read_image(path):
    path = path.decode()
    x = cv2.imread(path, cv2.IMREAD_COLOR)
    x = cv2.resize(x, (IMAGE_SIZE, IMAGE_SIZE))
    x = x/255.0
    return x

def read_mask(path):
    path = path.decode()
    x = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    x = cv2.resize(x, (IMAGE_SIZE, IMAGE_SIZE))
    x = x/255.0
    x = np.expand_dims(x, axis=-1)
    return x
```

Code snippet 1.2: Normalization of Input

2-Encoder:

The Encoder here consists of two main Blocks:

A- The convolution Block:
Which preforms two layers of convolutions between each layer we included:

A.1- Batch Normalization layer:

After a convolutional layer, it is common to apply batch normalization. This is because the convolutional layer produces a large number of filters which can have different scales and distributions, and batch normalization helps to normalize these filters, making the network more stable and easier to train by improving the gradient flow.

A.2- Dropout:

After a convolutional layer, it is common to apply dropout. This is because the convolutional layer produces a large number of filters, many of which are dependent on the provided data set only and not generalized to the unseen data, those filters can cause overfitting, and dropout helps to regularize the filters by randomly dropping some each time, making the network more robust and less prone to overfitting.

A.3- Leaky-ReLU Activation:

Leaky-ReLU is a variant of the Rectified Linear Unit (ReLU) activation function, which is commonly used in convolutional neural networks. Like ReLU, it allows the activations of a layer to be positive, but unlike ReLU, it allows a small negative slope for negative input values. This can help to improve the performance and stability of the network, particularly in deeper layers where the gradients can vanish or explode.

A.4- Residual block:

Here is a simple variable that we used to store the input in , and add it to the output of the first convolution, then we feed the resultant to the second convolution, this help prevent the gradient from vanishing.

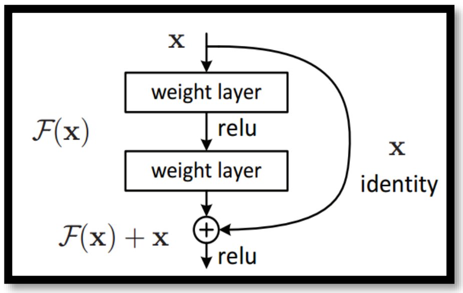
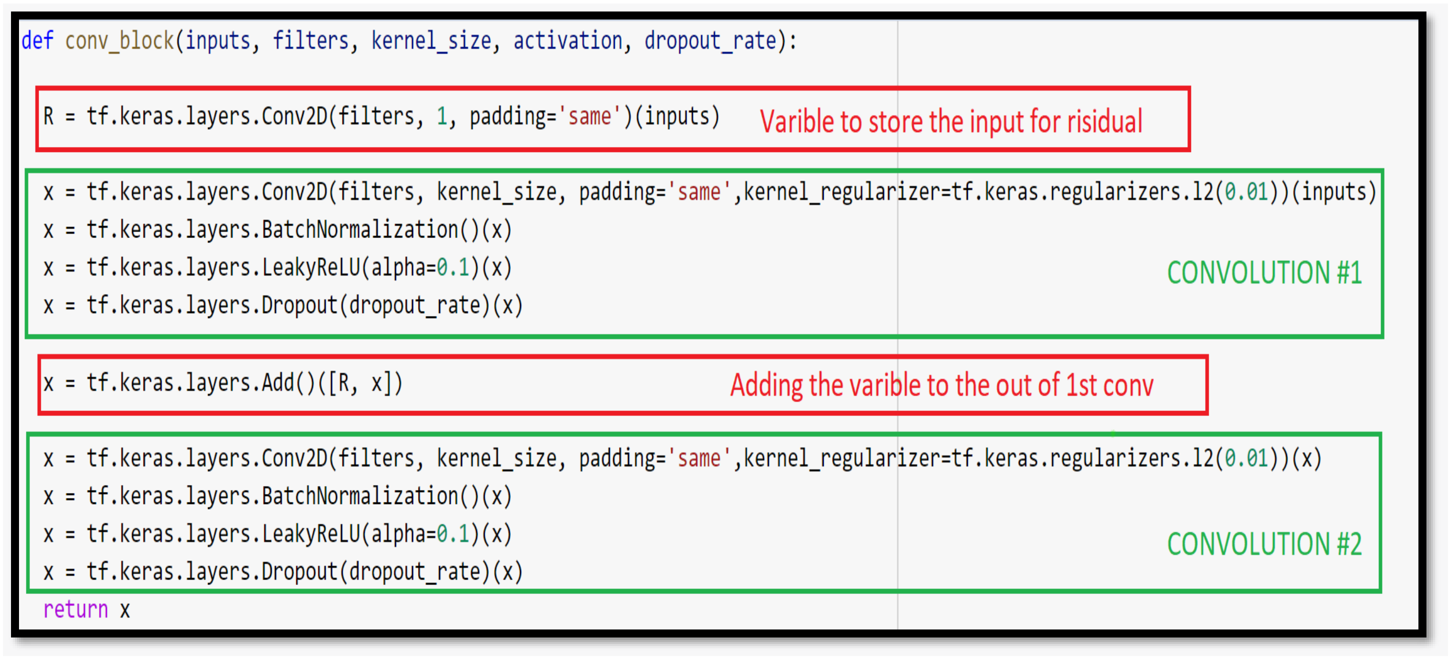

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 1.4: Residual Block

A.5- Weight decay:

As you can see the in-code snippet 1.3, we initialized our wights with kernel initializer L2, which will panelizes the model if wights are not regularized, and insure the model convergence, this also the reason on why our loss initially is a value so much greater than 1, because it panelizes the system hardly.

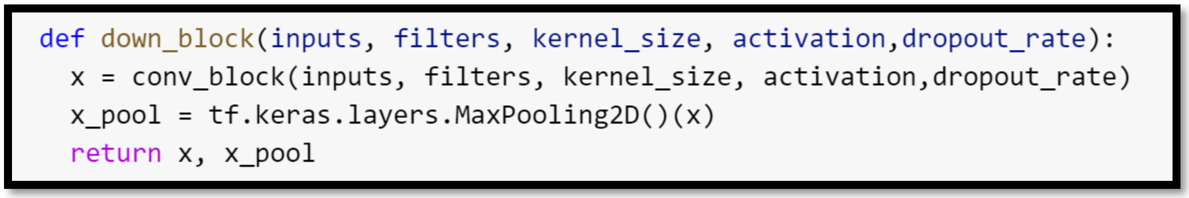
Weight decay can help to reduce overfitting and improve the generalization of the model by penalizing large weights, which can often be a sign of overfitting. It can also help to improve the stability and convergence of the optimization algorithm by smoothing out the landscape of the loss function.


Varible to store the input for risidual
CONVOLUTION #1
Adding the varible to the out of 1st conv
CONVOLUTION #2

```
def conv_block(inputs, filters, kernel_size, activation, dropout_rate):

  R = tf.keras.layers.Conv2D(filters, 1, padding='same')(inputs)

  x = tf.keras.layers.Conv2D(filters, kernel_size, padding='same',kernel_regularizer=tf.keras.regularizers.l2(0.01))(inputs)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.LeakyReLU(alpha=0.1)(x)
  x = tf.keras.layers.Dropout(dropout_rate)(x)

  x = tf.keras.layers.Add()([R, x])

  x = tf.keras.layers.Conv2D(filters, kernel_size, padding='same',kernel_regularizer=tf.keras.regularizers.l2(0.01))(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.LeakyReLU(alpha=0.1)(x)
  x = tf.keras.layers.Dropout(dropout_rate)(x)
  return x
```

Code snippet 1.3: Convolution Block

B- **The Down Scale Block:**
In this function we call the convolution function to preform convolution, then we apply max pooling on the results of the convolution which causing the network to downscale, this function returns a skip connection a pooled layer output. The skip connection will be used later in the upscale Block.



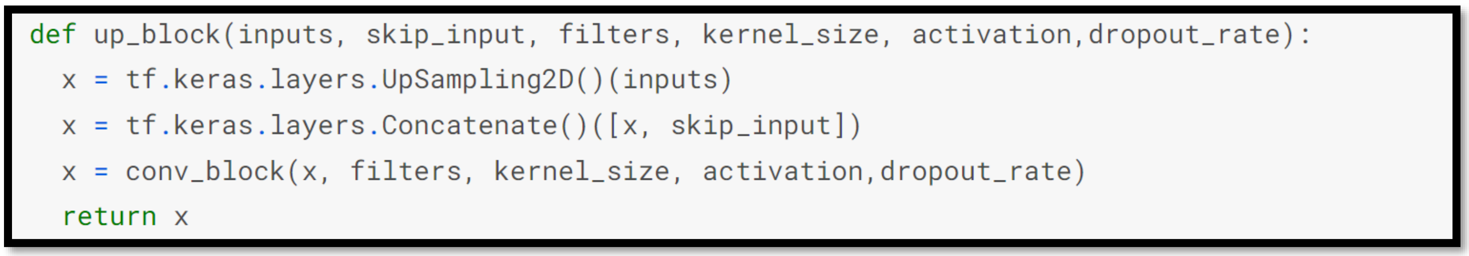
```
def down_block(inputs, filters, kernel_size, activation,dropout_rate):
  x = conv_block(inputs, filters, kernel_size, activation,dropout_rate)
  x_pool = tf.keras.layers.MaxPooling2D()(x)
  return x, x_pool
```

Code snippet 1.4: Down sampling

3-Decoder:

The Decoder is a block we use to up-sample the features and increase the spatial structure after the down-sample has reduced it to feature, this block preforms 3 functions:

A- Up- sampling: which is to reverse the pooling.
B- Concatenation: with the skip connections.
C- Convolution: here we use the filter sizes in reverse order.



```
def up_block(inputs, skip_input, filters, kernel_size, activation,dropout_rate):
  x = tf.keras.layers.UpSampling2D()(inputs)
  x = tf.keras.layers.Concatenate()([x, skip_input])
  x = conv_block(x, filters, kernel_size, activation,dropout_rate)
  return x
```

Code snippet 1.5: Up sampling

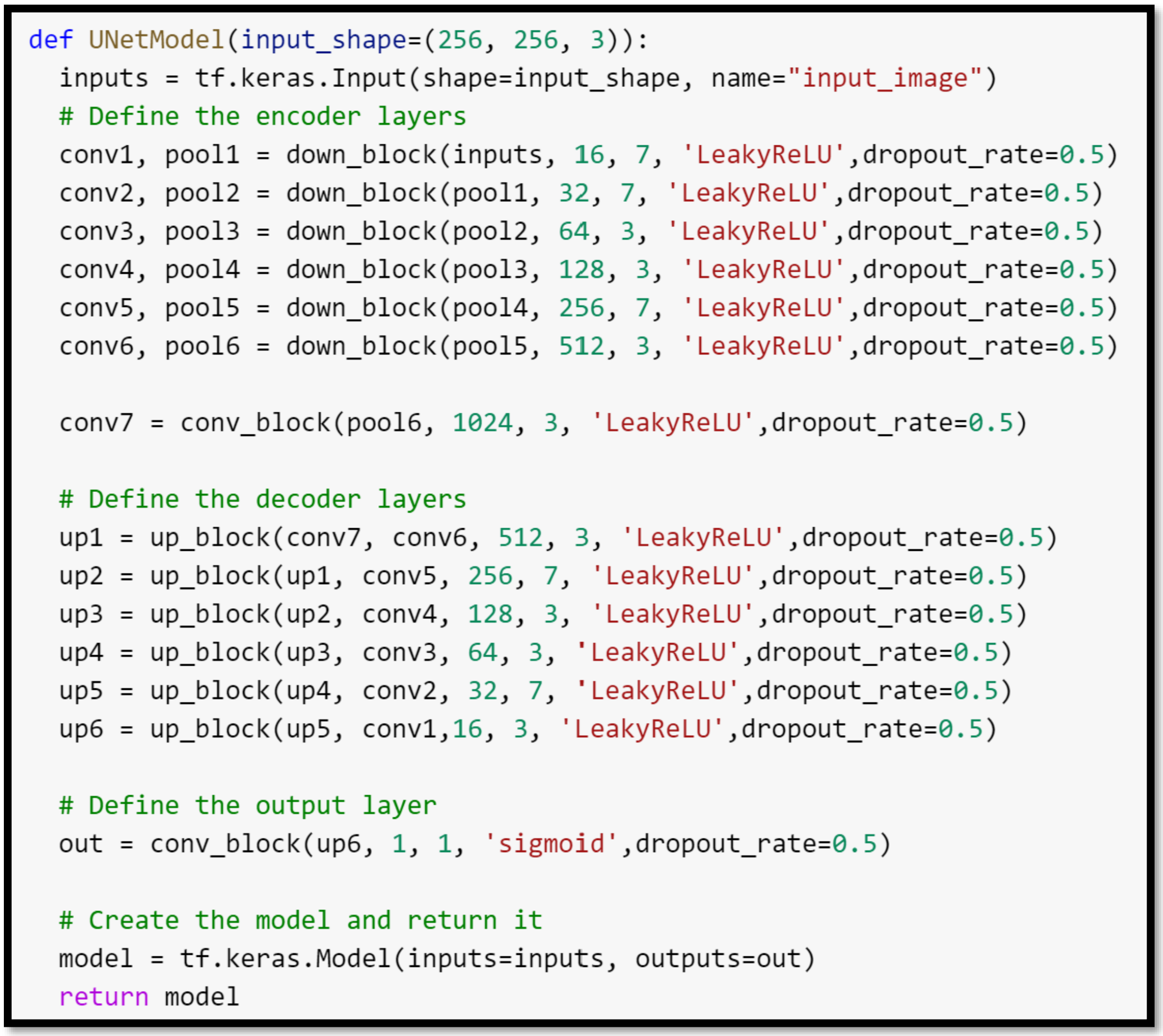
4-Model:

As was required in this project, The kernel sizes for each model is changed according to the Registration Numbers of the authors as following:

1- Reverse the registration number.
2- If any number is even, add 1 to convert to add.
3- Since each author has 6 digits, you will have6 kernels, 6 Encoder blocks and 6 Decoder blocks.

Author	Name	Registration	Reverse-Registration	Add 1 , if needed	Final Kernel sizes
Author 1	**Abdullah Abid**	**362376**	**673263**	**773373**	**7,7,3,3,7,3**
Author 2	**Khubaib Haidar**	**363126**	**621363**	**731373**	**7,3,1,3,7,3**
Author 3	**Faizan Faiz**	**362940**	**049263**	**159373**	**1,5,9,3,7,3**

Table 1.1: Number of parameter for each model



```python
def UNetModel(input_shape=(256, 256, 3)):
  inputs = tf.keras.Input(shape=input_shape, name="input_image")
  # Define the encoder layers
  conv1, pool1 = down_block(inputs, 16, 7, 'LeakyReLU',dropout_rate=0.5)
  conv2, pool2 = down_block(pool1, 32, 7, 'LeakyReLU',dropout_rate=0.5)
  conv3, pool3 = down_block(pool2, 64, 3, 'LeakyReLU',dropout_rate=0.5)
  conv4, pool4 = down_block(pool3, 128, 3, 'LeakyReLU',dropout_rate=0.5)
  conv5, pool5 = down_block(pool4, 256, 7, 'LeakyReLU',dropout_rate=0.5)
  conv6, pool6 = down_block(pool5, 512, 3, 'LeakyReLU',dropout_rate=0.5)

  conv7 = conv_block(pool6, 1024, 3, 'LeakyReLU',dropout_rate=0.5)

  # Define the decoder layers
  up1 = up_block(conv7, conv6, 512, 3, 'LeakyReLU',dropout_rate=0.5)
  up2 = up_block(up1, conv5, 256, 7, 'LeakyReLU',dropout_rate=0.5)
  up3 = up_block(up2, conv4, 128, 3, 'LeakyReLU',dropout_rate=0.5)
  up4 = up_block(up3, conv3, 64, 3, 'LeakyReLU',dropout_rate=0.5)
  up5 = up_block(up4, conv2, 32, 7, 'LeakyReLU',dropout_rate=0.5)
  up6 = up_block(up5, conv1,16, 3, 'LeakyReLU',dropout_rate=0.5)

  # Define the output layer
  out = conv_block(up6, 1, 1, 'sigmoid',dropout_rate=0.5)

  # Create the model and return it
  model = tf.keras.Model(inputs=inputs, outputs=out)
  return model
```

Code snippet 1.6: Model Function (Abdullah's model)

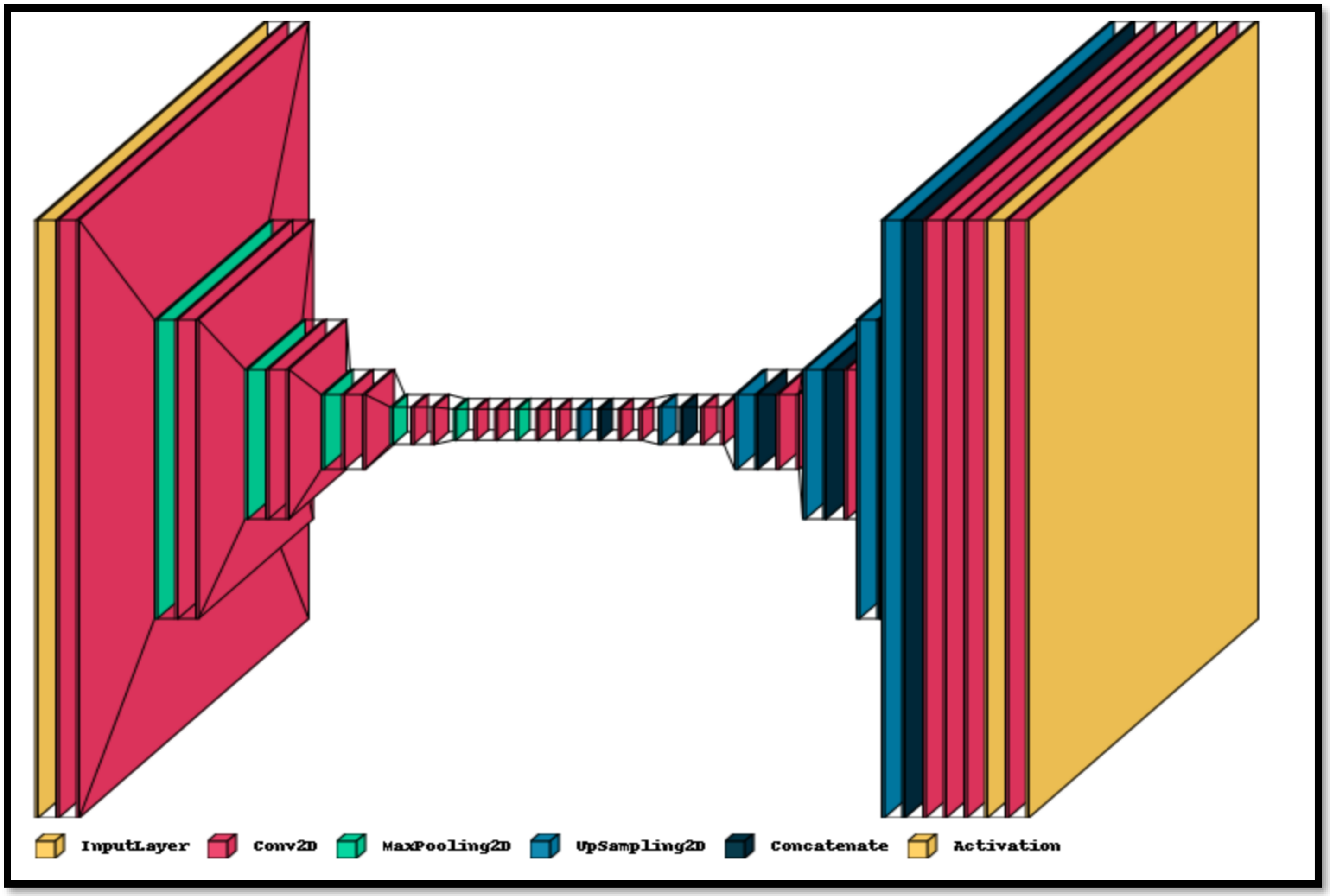
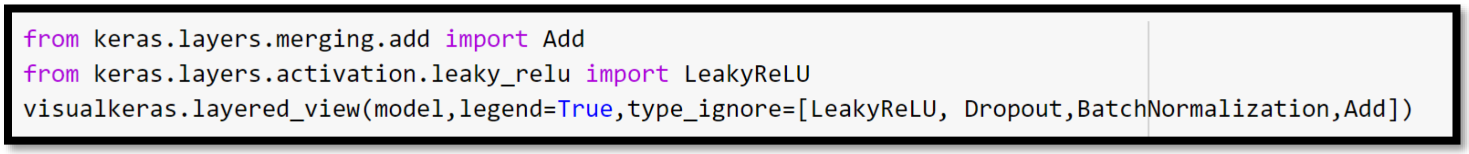


Figure 1.5: Model Architecture(obtained Using: visualkeras.layered_view(model))



```
from keras.layers.merging.add import Add
from keras.layers.activation.leaky_relu import LeakyReLU
visualkeras.layered_view(model,legend=True,type_ignore=[LeakyReLU, Dropout,BatchNormalization,Add])
```

Code snippet 1.7: printing the architecture showed in figure 1.5

The architecture in figure 1.5 was printed by the code snippet in 1.7, while visualizing this architecture I ignored some layers for the sake of consignment, you can see in the architecture its basically (itput then conv-conv-pool) until Up sampling which is (conv- conv-up sample)

Author name	Total parameter	Trainable	Non-trainable
Abdullah Abid	**46,179,083**	**46,166,919**	**12,164**
Khubaib Haidar	**45,773,579**	**45,761,415**	**12,164**
Faizan Faiz	**47,602,187**	**47,590,023**	**12,164**

Table 1.2: Number of parameter for each model

5-Loss function Utilized:

1- The Root Mean Squared Error (RMSE):

This is a common choice for tasks that involve predicting continuous values, such as depth estimation. The RMSE loss is defined as the square root of the mean of the squared differences between the predicted values and the true values. It is often used in regression tasks, where the goal is to minimize the difference between the predicted and true values.

```
def RMSE_loss(y_true, y_pred):
  return tf.keras.losses.mean_squared_error(y_true, y_pred)
```

Code snippet 1.8: RMSE loss

2- Dice loss:

This is a common loss function used in image segmentation tasks, where the goal is to predict a class or a mask for each pixel in an image. The Dice loss measures the overlap between the predicted and true masks.

```
smooth = 1e-15
def dice_coef(y_true, y_pred):
    y_true = tf.keras.layers.Flatten()(y_true)
    y_pred = tf.keras.layers.Flatten()(y_pred)
    intersection = tf.reduce_sum(y_true * y_pred)
    return (2. * intersection + smooth) / (tf.reduce_sum(y_true) + tf.reduce_sum(y_pred) + smooth)

def dice_loss(y_true, y_pred):
    return 1.0 - dice_coef(y_true, y_pred)
```

Code snippet 1.9: Dice Loss and Dice coefficient

3- IoU (Intersection over Union) loss:

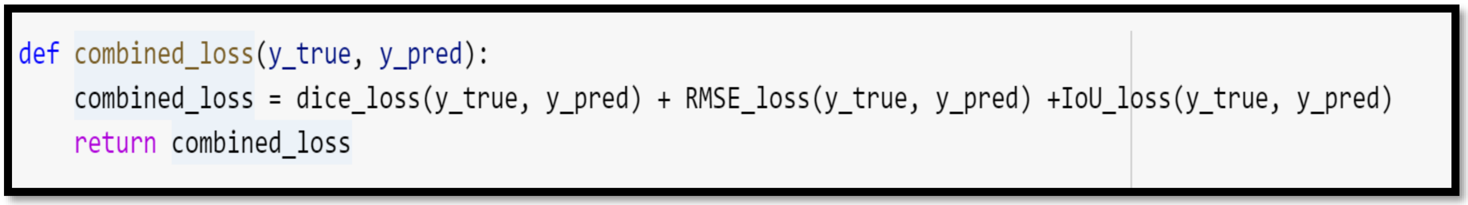
is a common metric used to evaluate the performance of image segmentation models. It measures the overlap between the predicted segmentation and the ground truth segmentation, and is calculated as the ratio of the intersection of the two to the union of the two.

```
def iou_loss(y_true, y_pred):
    y_true = tf.keras.backend.flatten(y_true)
    y_pred = tf.keras.backend.flatten(y_pred)
    intersection = tf.keras.backend.sum(y_true * y_pred)
    union = tf.keras.backend.sum(y_true) + tf.keras.backend.sum(y_pred) - intersection
    return 1 - (intersection + 1) / (union + 1)
```

Code snippet 1.10: Intersection over Union Loss

4- Combined loss:

The Authors thought about penalizing the model with respect to different losses , so the predicted mask would be as close as possible to the predicted mask, so they defined a Loss function that contains the Dice loss , RMSE loss, and the IoU loss



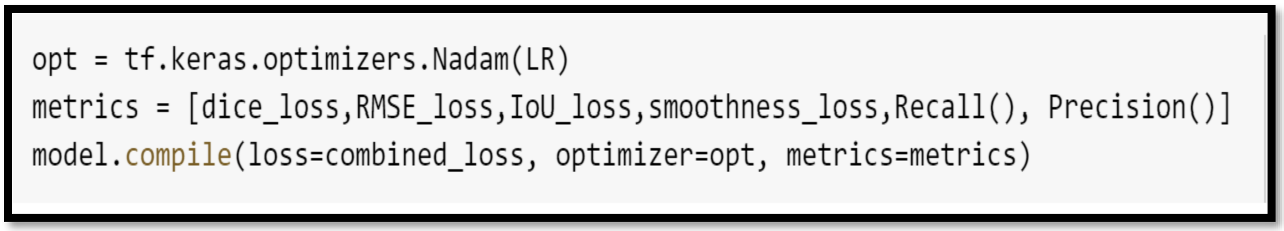
```python
def combined_loss(y_true, y_pred):
    combined_loss = dice_loss(y_true, y_pred) + RMSE_loss(y_true, y_pred) +IoU_loss(y_true, y_pred)
    return combined_loss
```

Code snippet 1.11: The combined loss used to compile the model

6-Optimization techniques:

Nadam (Nesterov Adam) is an optimization algorithm that is based on the Adam optimization algorithm. It is designed to combine the benefits of the Adam and Nesterov momentum optimization algorithms, and has been shown to perform well in a variety of machine learning tasks.

In the context of a depth estimation model, Nadam can be used to optimize the model's parameters in order to minimize the loss function. It does this by adapting the learning rate of the model based on the gradient of the loss function with respect to the model's parameters. This can help the model converge more quickly and accurately to a good solution.



```python
opt = tf.keras.optimizers.Nadam(LR)
metrics = [dice_loss,RMSE_loss,IoU_loss,smoothness_loss,Recall(), Precision()]
model.compile(loss=combined_loss, optimizer=opt, metrics=metrics)
```

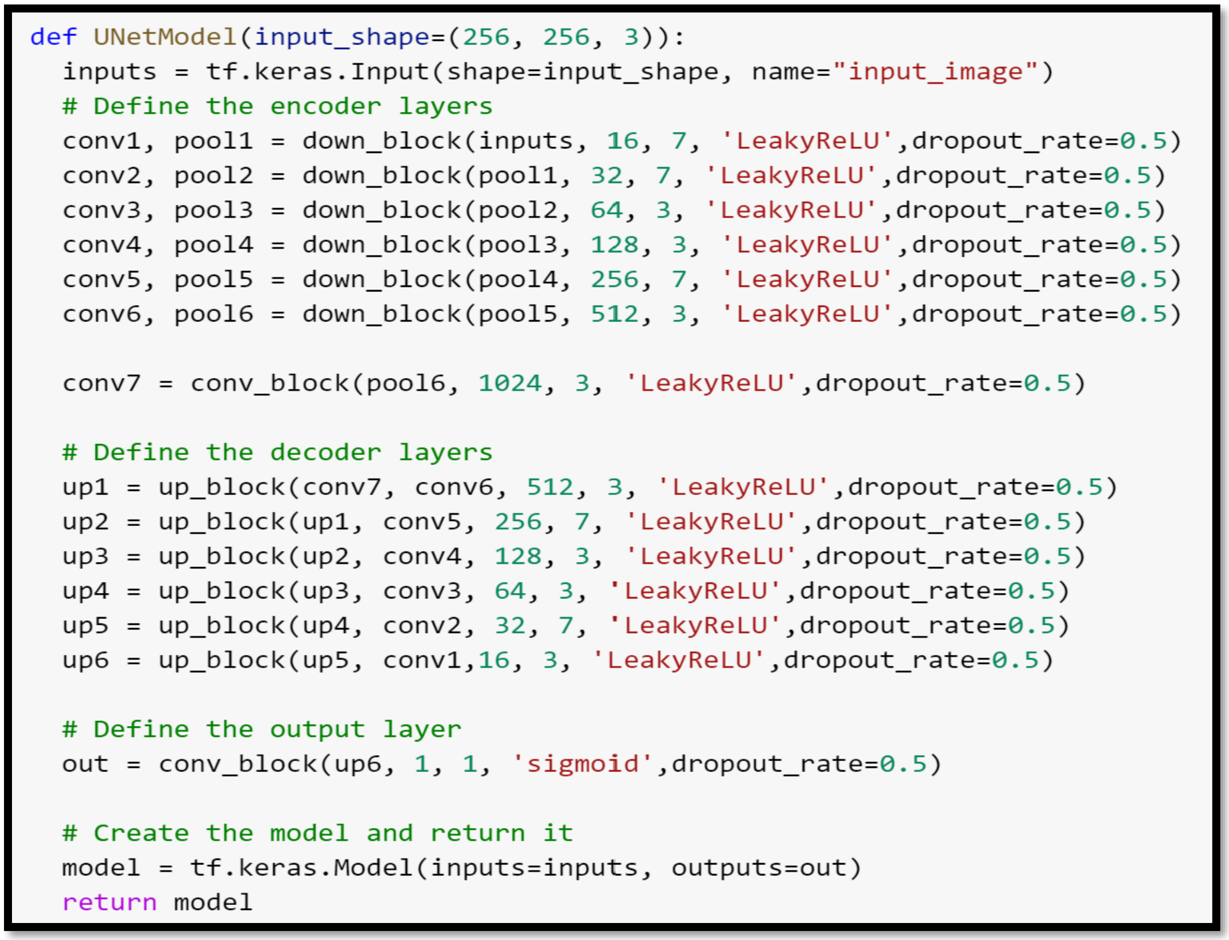
Code snippet 1.12: the optimizer and the compilation of the model

Also, as optimization, Batch normalization, Dropout and LeakyReLu as discussed in the Encoder above.

7- Results of each model:

First: Author One’s model “Abdullah Abid”:

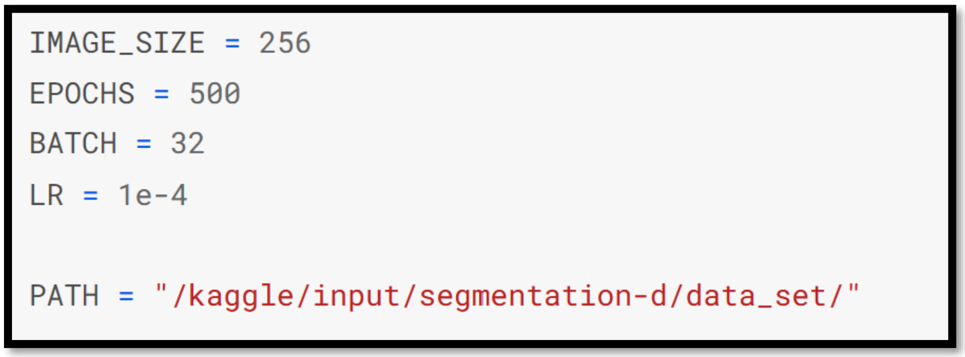
In this model we used the same kernel architecture as we were directed by the instructor, here are 6 conv layers with 6 Trans-conv layers:



```
def UNetModel(input_shape=(256, 256, 3)):
  inputs = tf.keras.Input(shape=input_shape, name="input_image")
  # Define the encoder layers
  conv1, pool1 = down_block(inputs, 16, 7, 'LeakyReLU',dropout_rate=0.5)
  conv2, pool2 = down_block(pool1, 32, 7, 'LeakyReLU',dropout_rate=0.5)
  conv3, pool3 = down_block(pool2, 64, 3, 'LeakyReLU',dropout_rate=0.5)
  conv4, pool4 = down_block(pool3, 128, 3, 'LeakyReLU',dropout_rate=0.5)
  conv5, pool5 = down_block(pool4, 256, 7, 'LeakyReLU',dropout_rate=0.5)
  conv6, pool6 = down_block(pool5, 512, 3, 'LeakyReLU',dropout_rate=0.5)

  conv7 = conv_block(pool6, 1024, 3, 'LeakyReLU',dropout_rate=0.5)

  # Define the decoder layers
  up1 = up_block(conv7, conv6, 512, 3, 'LeakyReLU',dropout_rate=0.5)
  up2 = up_block(up1, conv5, 256, 7, 'LeakyReLU',dropout_rate=0.5)
  up3 = up_block(up2, conv4, 128, 3, 'LeakyReLU',dropout_rate=0.5)
  up4 = up_block(up3, conv3, 64, 3, 'LeakyReLU',dropout_rate=0.5)
  up5 = up_block(up4, conv2, 32, 7, 'LeakyReLU',dropout_rate=0.5)
  up6 = up_block(up5, conv1,16, 3, 'LeakyReLU',dropout_rate=0.5)

  # Define the output layer
  out = conv_block(up6, 1, 1, 'sigmoid',dropout_rate=0.5)

  # Create the model and return it
  model = tf.keras.Model(inputs=inputs, outputs=out)
  return model
```

Code snippet 1.13: Abdullah’s Model

The code originally set to Run for 500 epochs, but due to the flatten output of validation loss the model went to Early Stopping, and stopped at `Epoch 56/500`



```
IMAGE_SIZE = 256
EPOCHS = 500
BATCH = 32
LR = 1e-4

PATH = "/kaggle/input/segmentation-d/data_set/"
```

Code snippet 1.14: Abdullah’s Model

Loss	Loss initial value at Train	Loss final value at Train	Loss initial value at Validation	Loss final value validation	Loss value at test
Combined Loss	9.7755	1.3153	2.3349	1.3127	1.3099
RMSE_loss	0.0656	0.0422	0.4236	0.0438	0.0438
dice_los	0.5825	0.5461	0.5541	0.5447	0.5426
IoU_loss	0.7360	0.7062	0.7128	0.7049	0.7032
recall	0.4123	0.3725	0.9928	0.3777	0.3818
precision	1.0000	1.0000	1.0000	1.0000	1.0000
accuracy	5.7845e-05	5.9885e-05	5.7538e-05	5.7514e-05	6.1980e-05

Table 1.3: Performance of the model-1 in train, valid and Test

Discussion of the results:

1- The model was set to run 500 epochs but "validation Accuracy" saturated after 56 epochs.
2- In table 1.3 you can see all the losses we used are decrementing, which means that the model is learning with respect to multiple parameters like regression value of the pixel, dice loss, and intersection loss.
3- You may notice that the accuracy is too low, that is because we used our own customized Loss function "Combined loss", this function is a combination of multiple losses, that's why you notice that the Combined loss in table 1.3 is more than one.

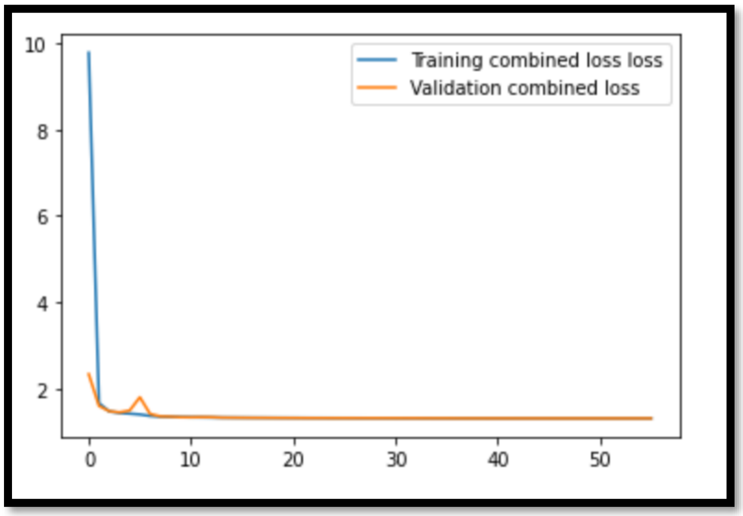


Figure 1.6: over all curve of loss s iteration (Train x Valid)

4- It maybe difficult to see in this curve , but the over all gap between training loss and validation loss is so small , which means that the model is not overfitting , because the larger the gap is the more error you get in the unseen data, But from table 1.3 and figure 1.6 you can see the model preforms just as well in unseen data “validation” as it does in Training.

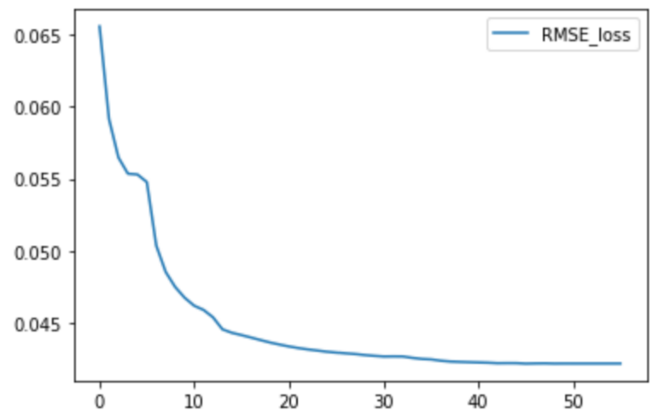


Figure 1.7: RMSE loss vs Iterations

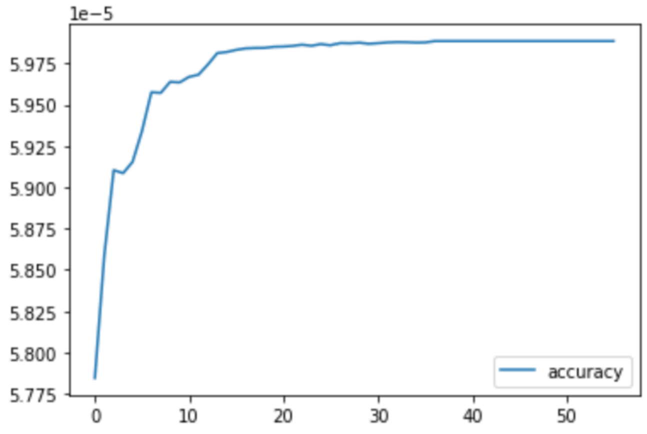


Figure 1.8: Accuracy vs Iterations

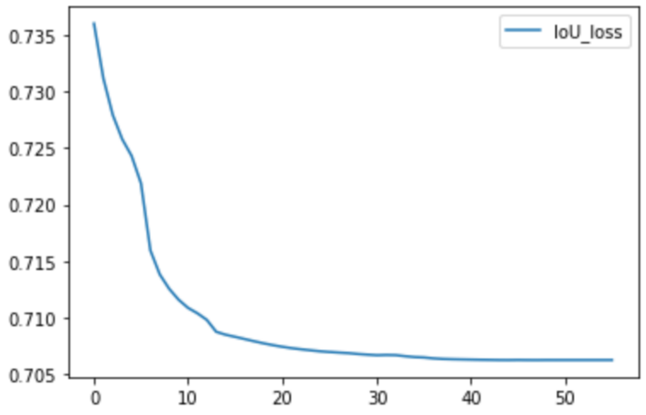


Figure 1.9: IoU loss vs Iteration

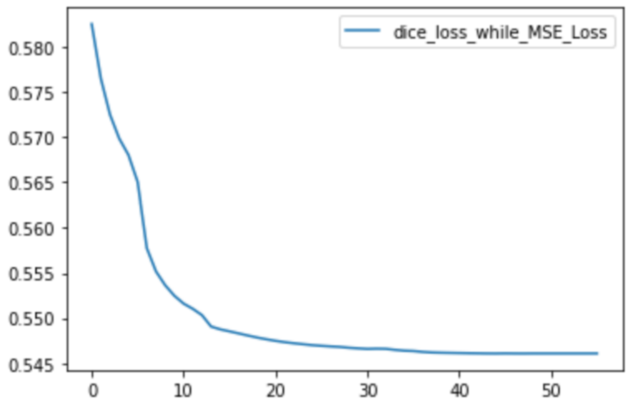


Figure 1.10: Dice loss vs Iteration

5- You notice in all the Losses curves above that all losses are coming down and saturating at some point, except the accuracy curve it rises up as it should be and behaving in opposition to the loss curve.

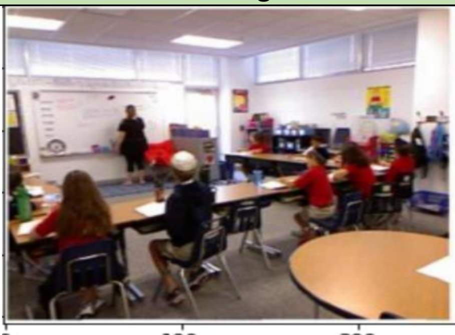
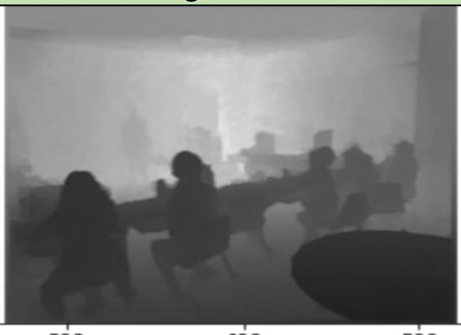
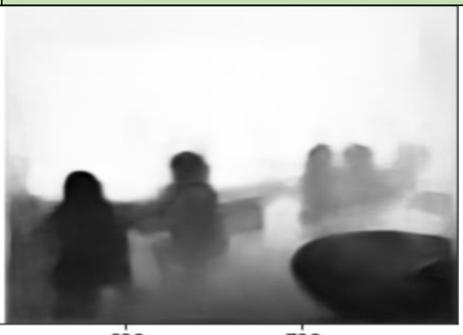
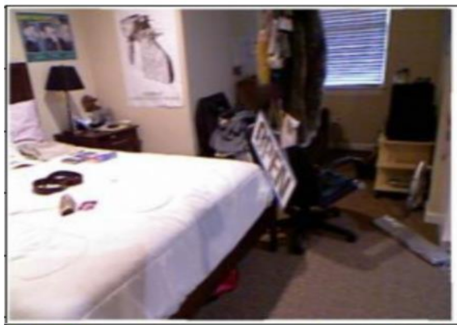
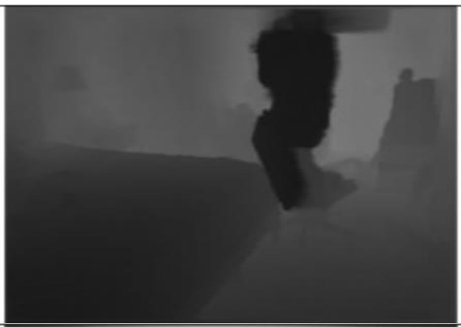
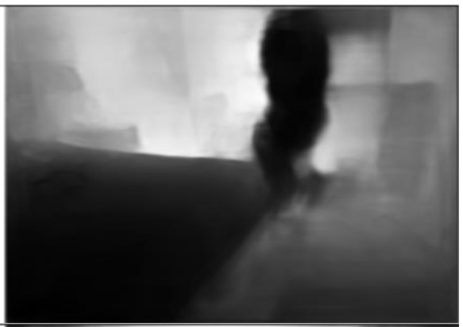
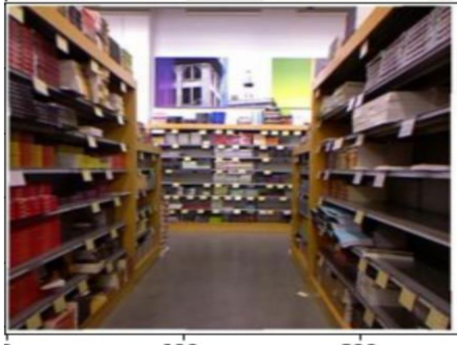
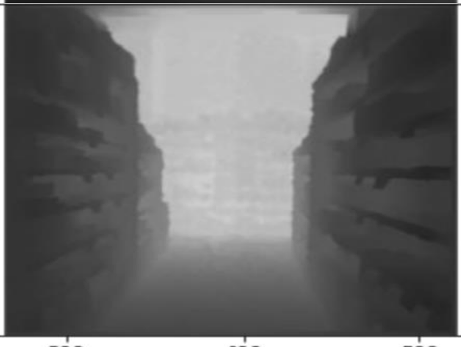
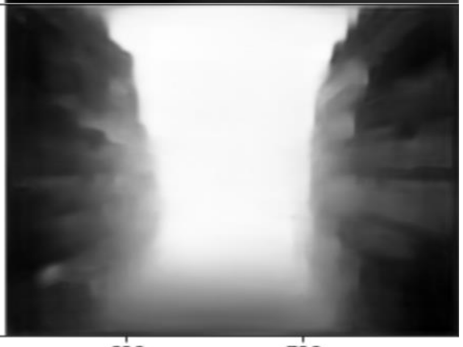
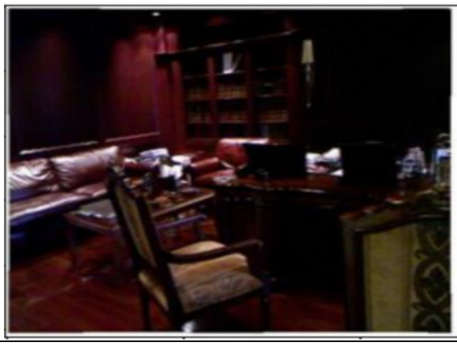
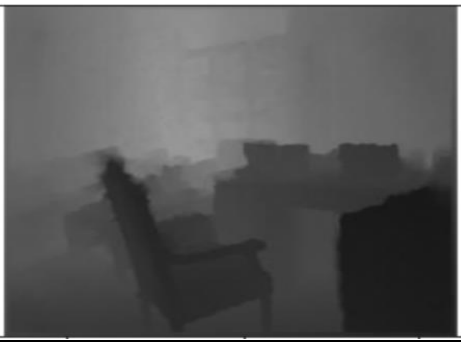
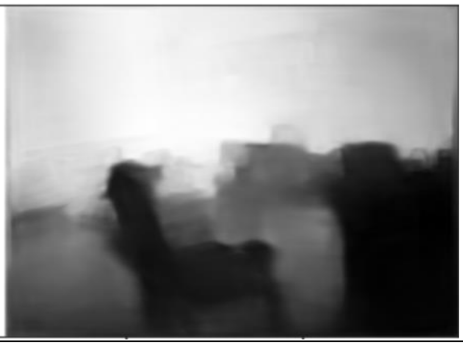
6- Notice that the Dice loss and the IoU loss have almost the same curve, because the logic behind both curves is the same basically, which is how much of the predicted mask matches the original mask.

For Future improvement:

Although Abdullah’s model preformed good, but maybe the result could be improved even further if we added more layers on encoder and decoder, increased the number of filters, increased the training data, tuned the hyperparameter even further.

In the following section, you will see image results of the depth estimation model of Abdullah’s model, seeing those images you will get some intuition on how the model were able to detect a very fine feature from a given RGB image and predict such a regression “grayscale” mask.

Image results of Model -1 (Abdullah Abid):

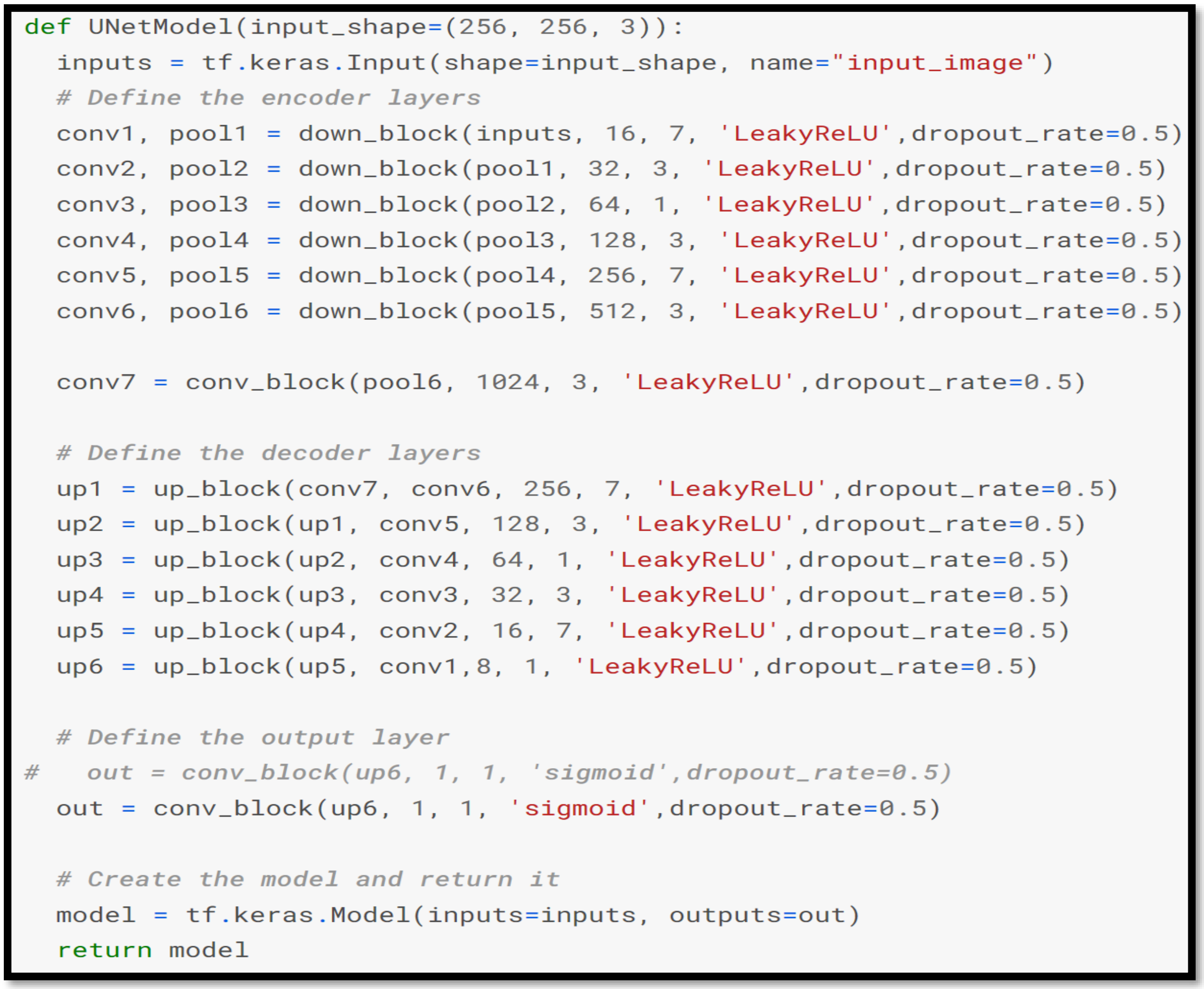
RGB Image	Original Mask	Predicted Mask
		
		
		
		

- Those images may not look clear on printing, please refer to the soft copy of the paper.

7- Results of each model: (Continued)

Third: Author Two’s model “Khubaib Haidar”:

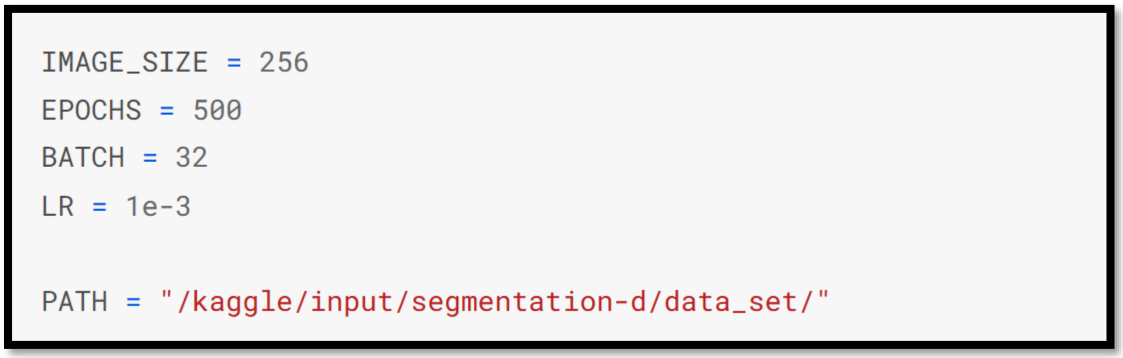
In this model we used the same kernel architecture as we were directed by the instructor, here are 6 conv layers with 6 Trans-conv layers:



```python
def UNetModel(input_shape=(256, 256, 3)):
  inputs = tf.keras.Input(shape=input_shape, name="input_image")
  # Define the encoder layers
  conv1, pool1 = down_block(inputs, 16, 7, 'LeakyReLU',dropout_rate=0.5)
  conv2, pool2 = down_block(pool1, 32, 3, 'LeakyReLU',dropout_rate=0.5)
  conv3, pool3 = down_block(pool2, 64, 1, 'LeakyReLU',dropout_rate=0.5)
  conv4, pool4 = down_block(pool3, 128, 3, 'LeakyReLU',dropout_rate=0.5)
  conv5, pool5 = down_block(pool4, 256, 7, 'LeakyReLU',dropout_rate=0.5)
  conv6, pool6 = down_block(pool5, 512, 3, 'LeakyReLU',dropout_rate=0.5)

  conv7 = conv_block(pool6, 1024, 3, 'LeakyReLU',dropout_rate=0.5)

  # Define the decoder layers
  up1 = up_block(conv7, conv6, 256, 7, 'LeakyReLU',dropout_rate=0.5)
  up2 = up_block(up1, conv5, 128, 3, 'LeakyReLU',dropout_rate=0.5)
  up3 = up_block(up2, conv4, 64, 1, 'LeakyReLU',dropout_rate=0.5)
  up4 = up_block(up3, conv3, 32, 3, 'LeakyReLU',dropout_rate=0.5)
  up5 = up_block(up4, conv2, 16, 7, 'LeakyReLU',dropout_rate=0.5)
  up6 = up_block(up5, conv1,8, 1, 'LeakyReLU',dropout_rate=0.5)

  # Define the output layer
#   out = conv_block(up6, 1, 1, 'sigmoid',dropout_rate=0.5)
  out = conv_block(up6, 1, 1, 'sigmoid',dropout_rate=0.5)

  # Create the model and return it
  model = tf.keras.Model(inputs=inputs, outputs=out)
  return model
```

Code snippet 1.15: Khubaib’s Model

The code originally set to Run for 500 epochs, but due to the flatten output of validation loss the model went to Early Stopping, and stopped at `Epoch 68/500`



```python
IMAGE_SIZE = 256
EPOCHS = 500
BATCH = 32
LR = 1e-3


PATH = "/kaggle/input/segmentation-d/data_set/"
```

Code snippet 1.16: Khubaib’s Model

Loss	Loss initial value at Train	Loss final value at Train	Loss initial value at Validation	Loss final value validation	Loss value at test
Combined Loss	2.5935	1.2900	1.5431	1.2876	1.2922
RMSE_loss	0.0803	0.0501	0.0812	0.0497	0.0500
dice_los	0.5990	0.5359	0.6198	0.5356	0.5372
IoU_loss	0.7491	0.6976	0.7652	0.6973	0.6985
recall	0.4709	0.3717	0.9558	0.3717	0.3709
precision	1.0000	1.0000	1.0000	1.0000	1.0000
accuracy	5.4186e-05	5.9659e-05	5.6238e-05	5.6009e-05	6.5086e-05

Table 1.4: Performance of the model-2 in train, valid and Test

Discussion of the results:

1- The model was set to run 500 epochs but "validation Accuracy" saturated after 68 epochs.
2- In table 1.4 you can see all the losses we used are decrementing, which means that the model is learning but the accuracy is very slowly increasing, in the case of validation decreasing a little bit.

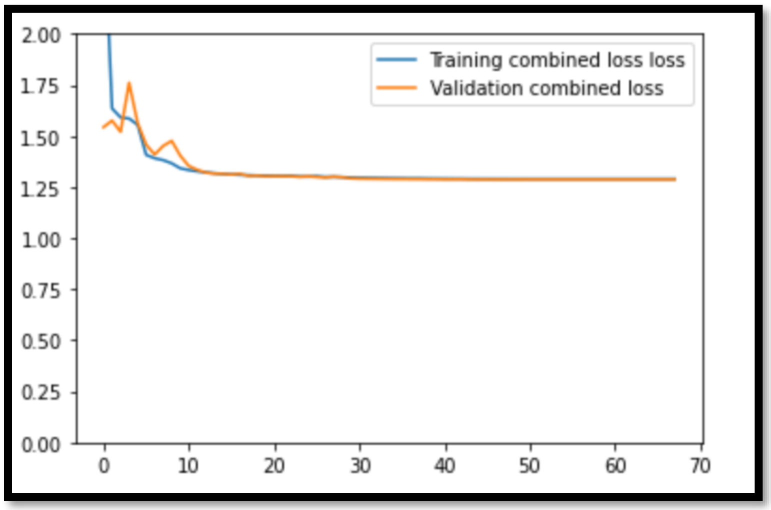


Figure 1.11: over all curve of loss s iteration (Train x Valid)

3- This curve is a bit zoomed in to clarify that there is almost no gap , which means that the model is not overfitting , because the larger the gap is the more error you get in the unseen data. Which kinda explains why the model is doing even better at test time.

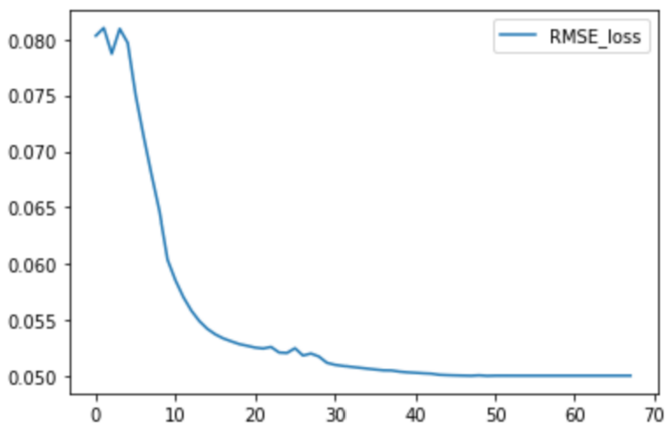


Figure 1.12: RMSE loss vs Iterations

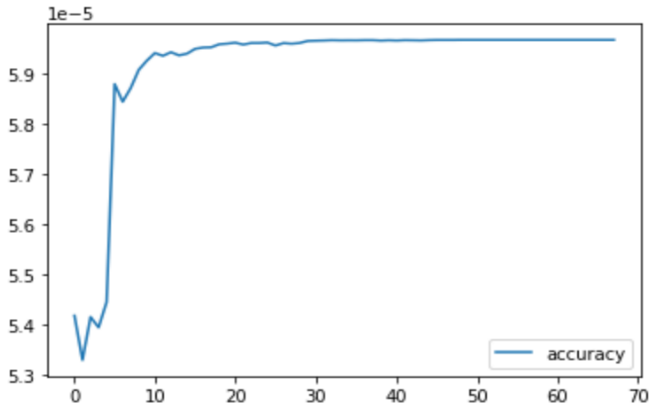


Figure 1.13: Accuracy vs Iterations

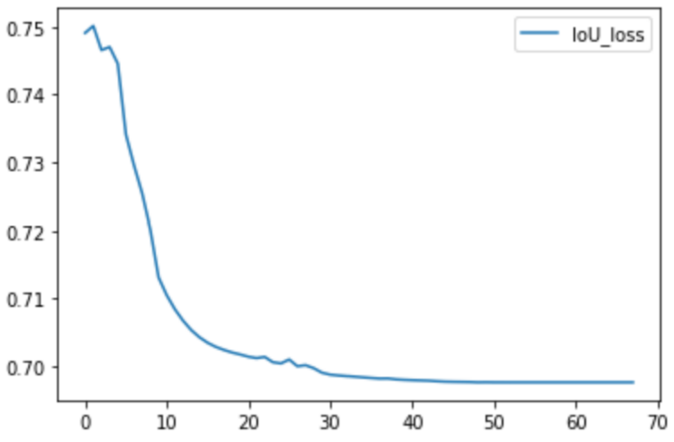


Figure 1.14: IoU loss vs Iteration

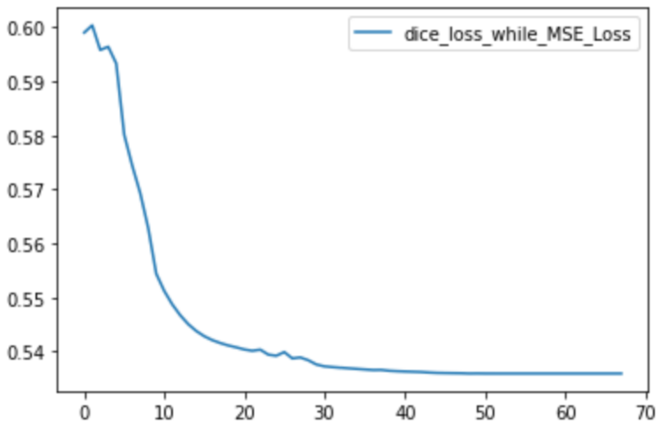


Figure 1.15: Dice loss vs Iteration

4- You notice in all the Losses curves above that all losses are coming down and saturating at some point, except the accuracy curve it rises up as it should be and behaving in opposition to the loss curve.

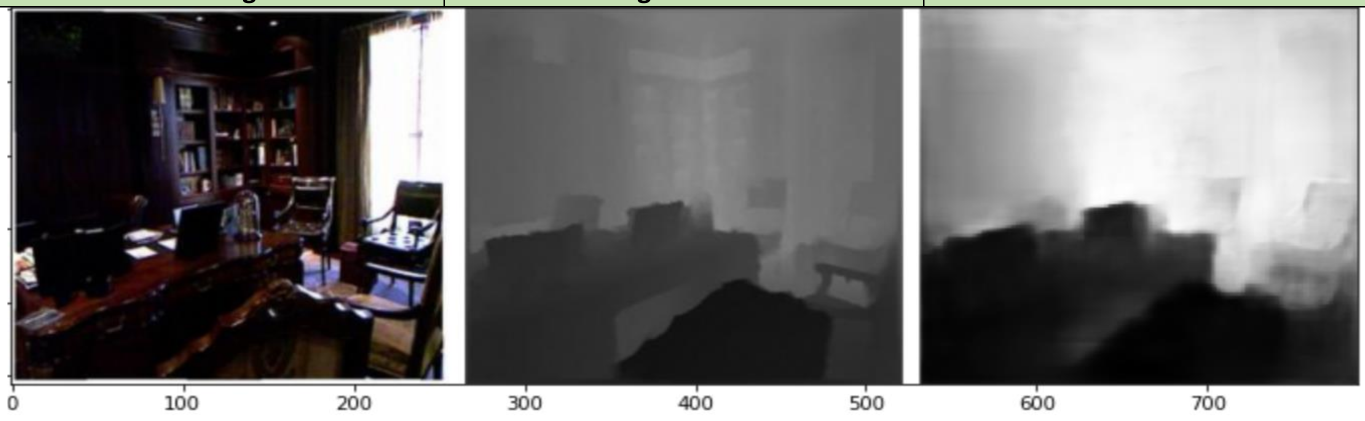
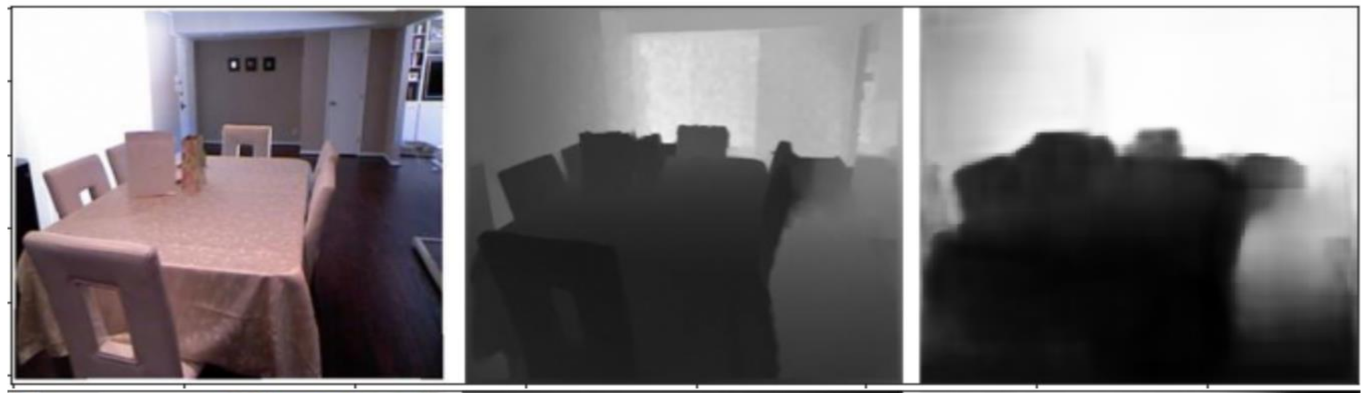
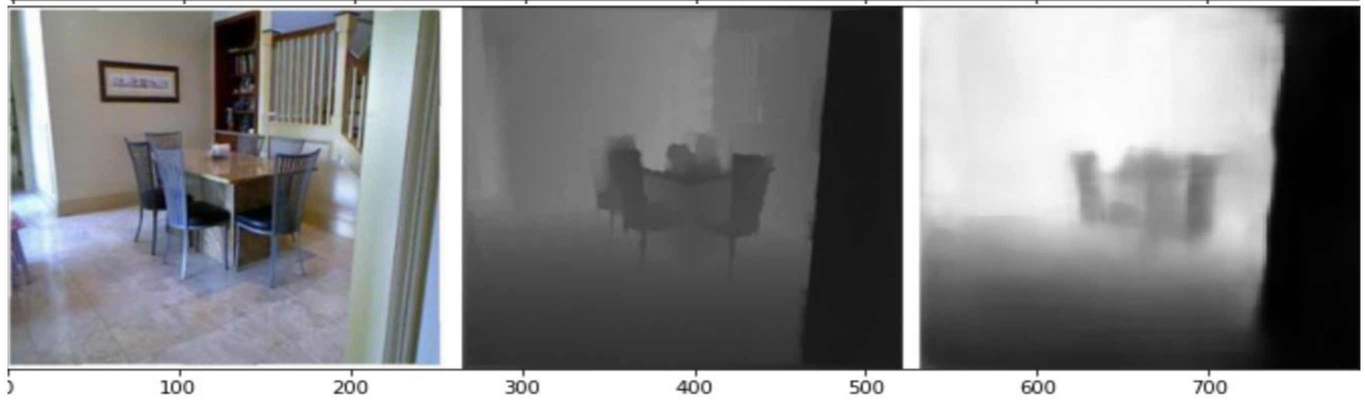
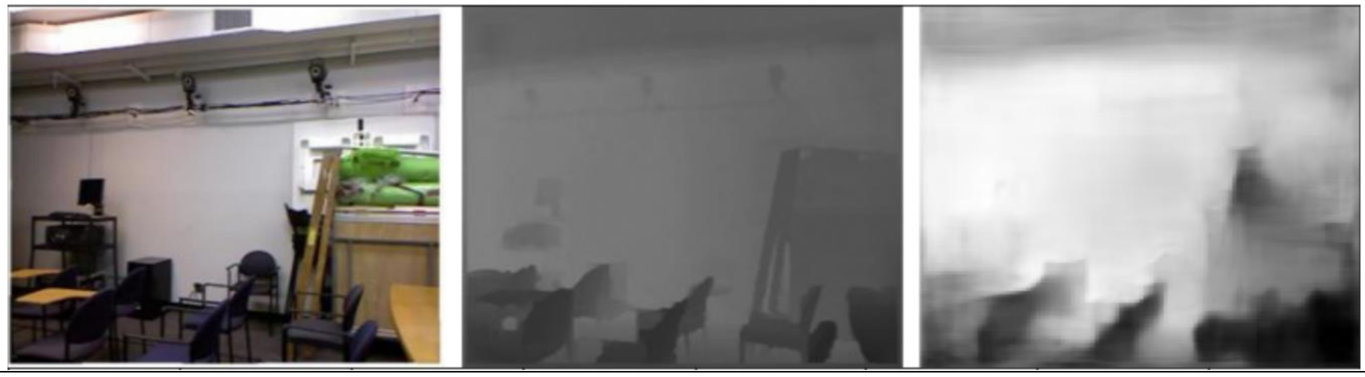
5- Notice that the Dice loss and the IoU loss have almost the same curve, because the logic behind both curves is the same basically, which is how much of the predicted mask matches the original mask.

For Future improvement:

Although we changed the learning Rate and tackled the learning a little bit, the Khubaib model preformed well but did show extraordinary results over Abdullah's model as we hoped, but maybe the result could be improved even further if we added more layers on encoder and decoder, increased the number of filters, increased the training data, tuned the hyperparameter even further.

In the following section, you will see image results of the depth estimation model of Khubaib model, seeing those images you will get some intuition on how the model were able to detect a very fine feature from a given RGB image and predict such a regression "grayscale" mask.

Image results of Model -2 (Khubaib Haidar):

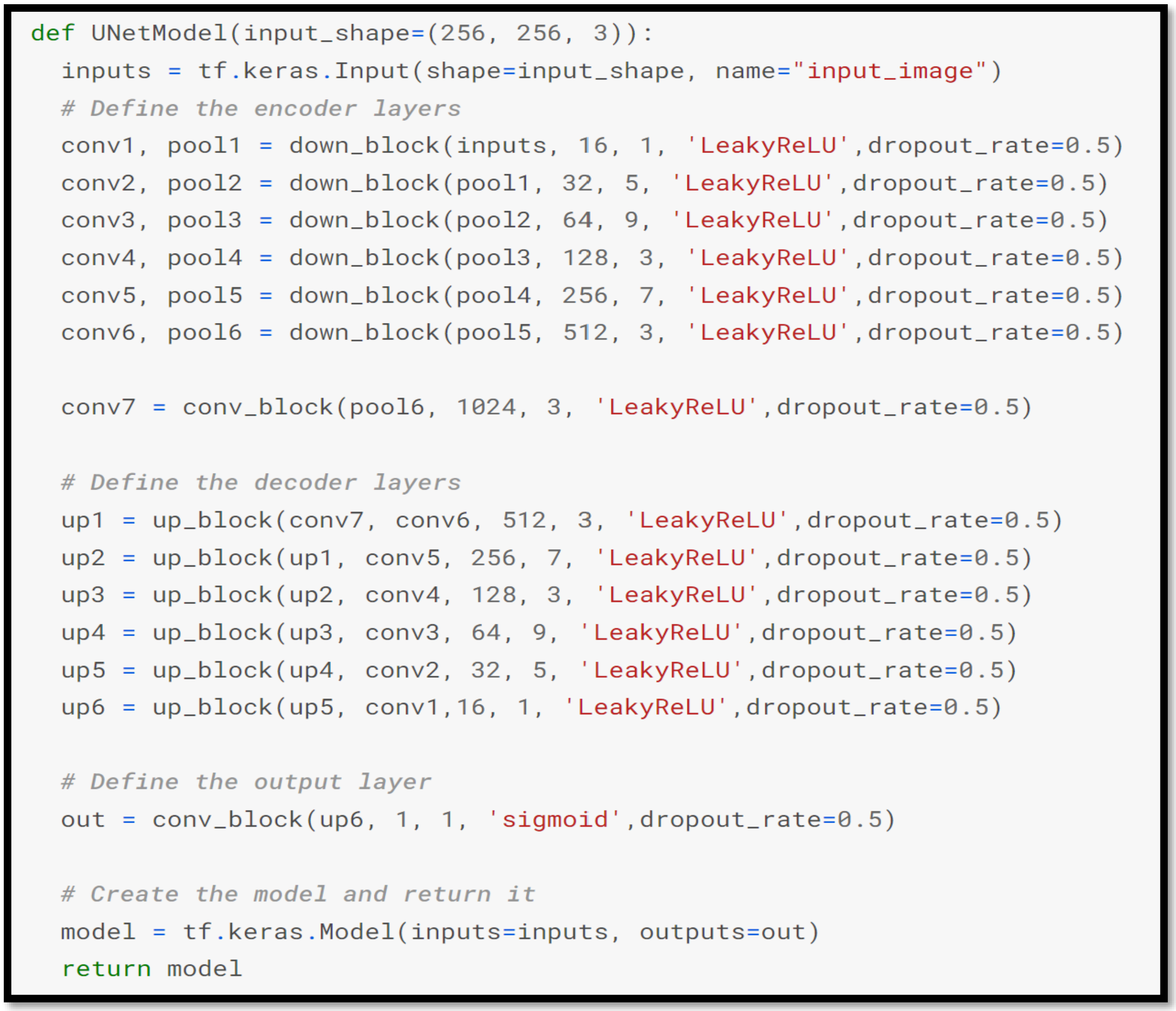
RGB Image	Original Mask	Predicted Mask
		
		
		
		

- Those images may not look clear on printing, please refer to the soft copy of the paper.

7- Results of each model: (Continued)

Third: Author Three’s model “Faizaan Faiz”:

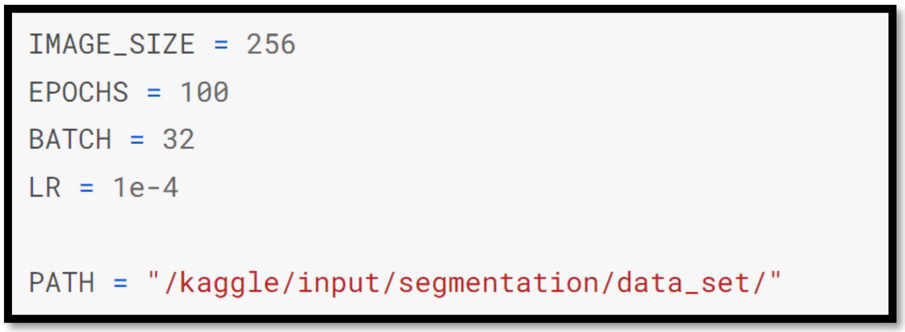
In this model we used the same kernel architecture as we were directed by the instructor, here are 6 conv layers with 6 Trans-conv layers:



```python
def UNetModel(input_shape=(256, 256, 3)):
  inputs = tf.keras.Input(shape=input_shape, name="input_image")
  # Define the encoder layers
  conv1, pool1 = down_block(inputs, 16, 1, 'LeakyReLU',dropout_rate=0.5)
  conv2, pool2 = down_block(pool1, 32, 5, 'LeakyReLU',dropout_rate=0.5)
  conv3, pool3 = down_block(pool2, 64, 9, 'LeakyReLU',dropout_rate=0.5)
  conv4, pool4 = down_block(pool3, 128, 3, 'LeakyReLU',dropout_rate=0.5)
  conv5, pool5 = down_block(pool4, 256, 7, 'LeakyReLU',dropout_rate=0.5)
  conv6, pool6 = down_block(pool5, 512, 3, 'LeakyReLU',dropout_rate=0.5)

  conv7 = conv_block(pool6, 1024, 3, 'LeakyReLU',dropout_rate=0.5)

  # Define the decoder layers
  up1 = up_block(conv7, conv6, 512, 3, 'LeakyReLU',dropout_rate=0.5)
  up2 = up_block(up1, conv5, 256, 7, 'LeakyReLU',dropout_rate=0.5)
  up3 = up_block(up2, conv4, 128, 3, 'LeakyReLU',dropout_rate=0.5)
  up4 = up_block(up3, conv3, 64, 9, 'LeakyReLU',dropout_rate=0.5)
  up5 = up_block(up4, conv2, 32, 5, 'LeakyReLU',dropout_rate=0.5)
  up6 = up_block(up5, conv1,16, 1, 'LeakyReLU',dropout_rate=0.5)

  # Define the output layer
  out = conv_block(up6, 1, 1, 'sigmoid',dropout_rate=0.5)

  # Create the model and return it
  model = tf.keras.Model(inputs=inputs, outputs=out)
  return model
```

Code snippet 1.17: Fizan’s Model

The code originally set to Run for 100 epochs, but due to the flatten output of validation loss the model went to Early Stopping, and stopped at `Epoch 43/100`.



```python
IMAGE_SIZE = 256
EPOCHS = 100
BATCH = 32
LR = 1e-4

PATH = "/kaggle/input/segmentation/data_set/"
```

Code snippet 1.18: Fizan’s Model

Loss	Loss initial value at Train	Loss final value at Train	Loss initial value at Validation	Loss final value validation	Loss value at test
Combined Loss	9.2615	1.2869	2.1075	1.2873	1.2823
HUBER_Loss	0.0375	0.0363	0.0182	0.0373	0.0368
dice_los	0.5808	0.5344	0.6572	0.5337	0.5315
IoU_loss	0.7346	0.6963	0.7931	0.6958	0.6939
recall	0.4804	0.4568	0.0056	0.4633	0.4613
precision	1.0000	1.0000	1.0000	1.0000	1.0000

Table 1.5: Performance of the model-3 in train, valid and Test

Discussion of the results:

1- You may have noticed that this model uses a different loss function than the RMSE, this was an initiative from the authors in attempt to experiment to improve the results.

2- The Huber loss is a loss function that is used in regression problems. It is a combination of the mean squared error (MSE) and the mean absolute error (MAE) loss functions. The Huber loss function is more robust to outliers than the MSE loss function, because it is less sensitive to large errors. However, it is also less sensitive to small errors than the MAE loss function.

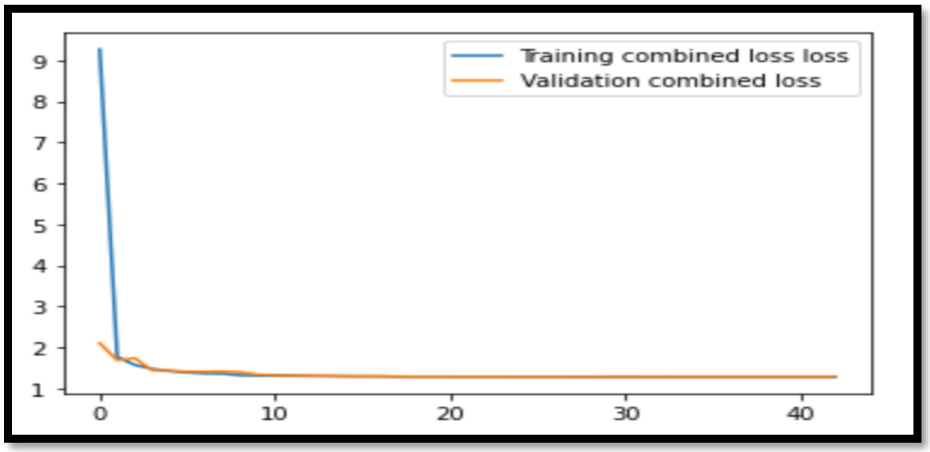


Figure 1.16: over all curve of loss s iteration (Train x Valid)

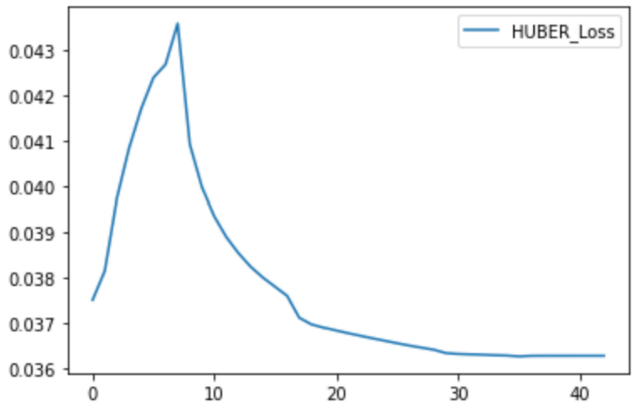
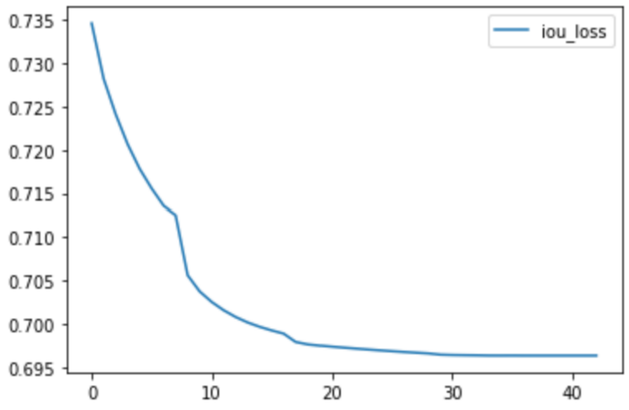


Figure 1.17: HUBER loss vs Iterations



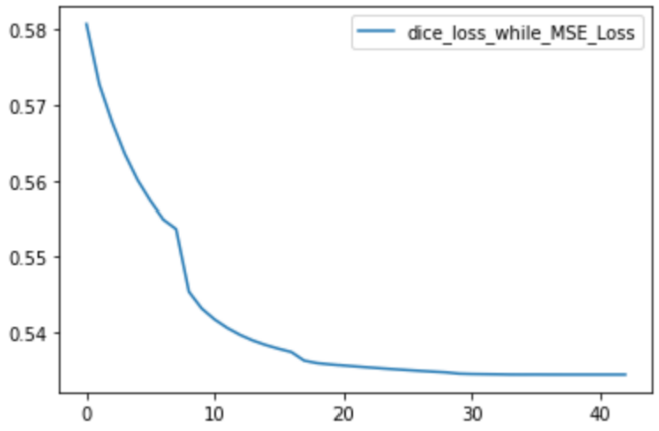


Figure 1.18: IoU loss vs Iteration

Figure 1.19: Dice loss vs Iteration

3- In this model the Authors forgot to plot Accuracy plot, but since the accuracy behave in opposite to the loss function you can imagine what the graph would look like. It is basically increasing.
4- You notice that the behavior of this model is resembles the Model_1 model except for the HUBER loss which makes since, since it's a different function, You can notice how the Huber loss varies as it says sensitivity to errors.

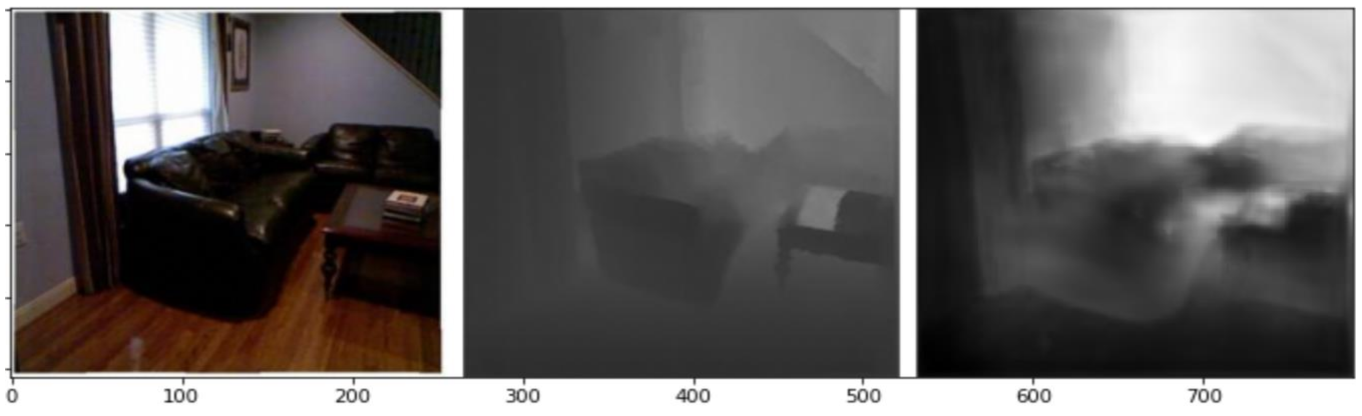
For Future improvement:

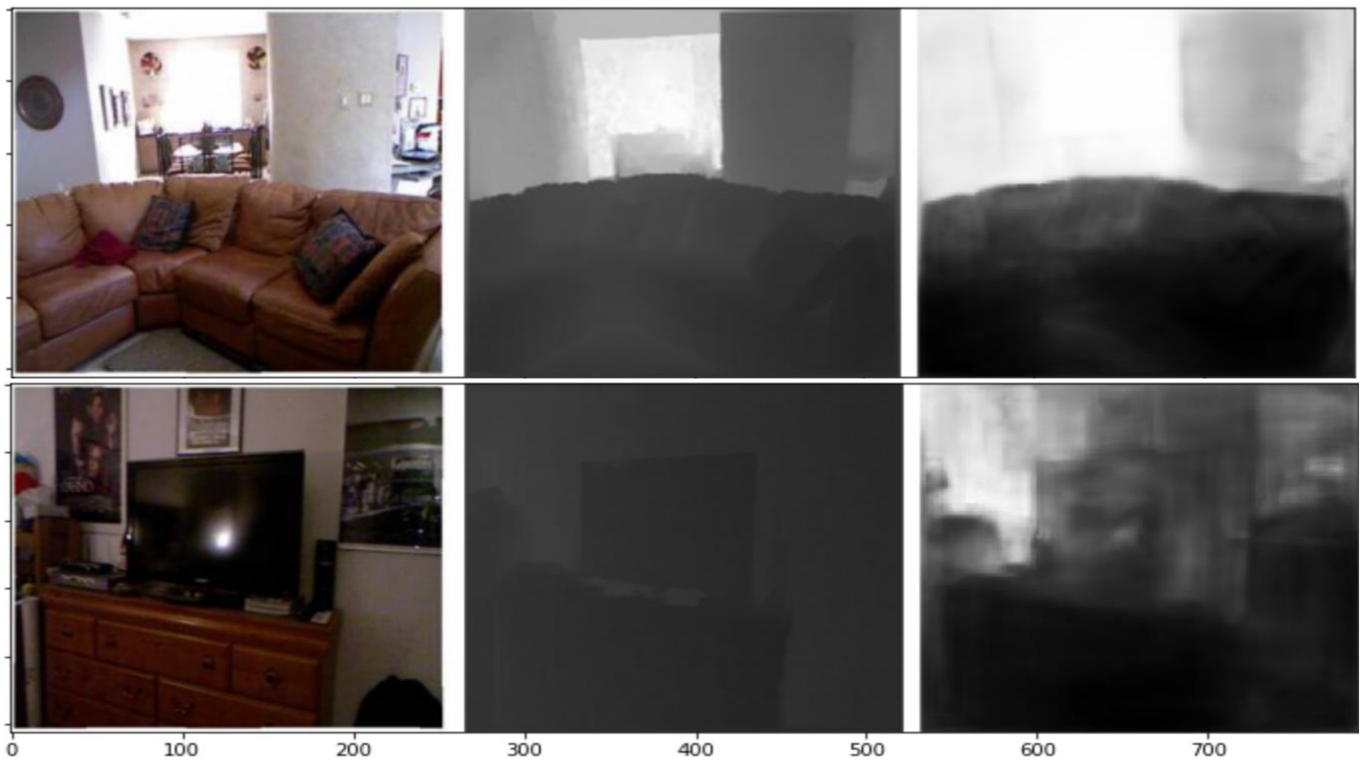
Although Faizaan's model preformed good, it achieved slightly better Dice-loss than Abdullah's model , but maybe the result could be improved even further if we added more layers on encoder and decoder, increased the number of filters, increased the training data, tuned the hyperparameter even further.

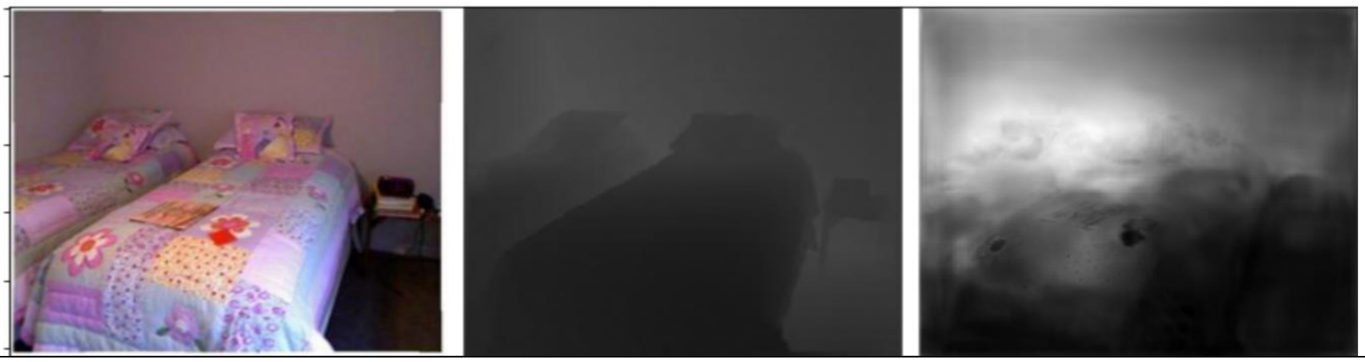
In the following section, you will see image results of the depth estimation model of Faizaan's model, seeing those images you will get some intuition on how the model were able to detect a very fine feature from a given RGB image and predict such a regression "grayscale" mask.

Image results of Model -3 (Faizaan Faiz):

RGB Image	Original Mask	Predicted Mask







- Those images may not look clear on printing, please refer to the soft copy of the paper.

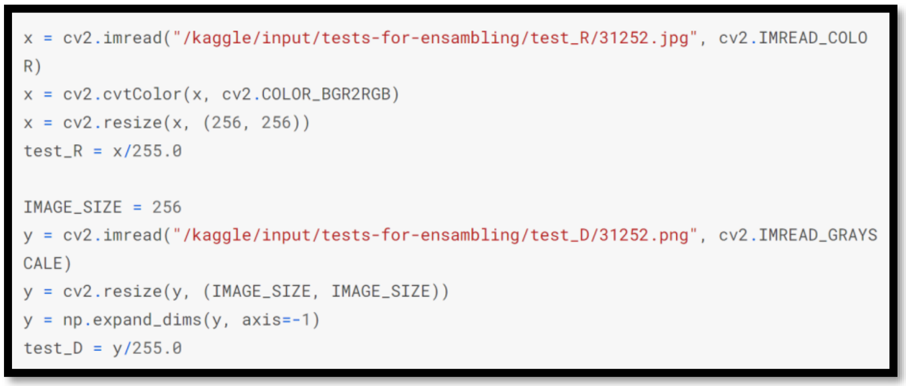
8-Model ensembling:

Model ensembling is a machine learning technique that combines the predictions of multiple models to achieve improved performance. The basic idea is that by combining the predictions of several models, we can reduce the variance and improve the generalizability of the final prediction.

Since all three models preformed pretty well, and there results are almost matching , the authors decided to preform weighted average model ensembling method, the ensembling method will help darken the output mask bringing it closer to the actual mask which is shown to be usually darker here.

To do model ensembling we preformed those steps:

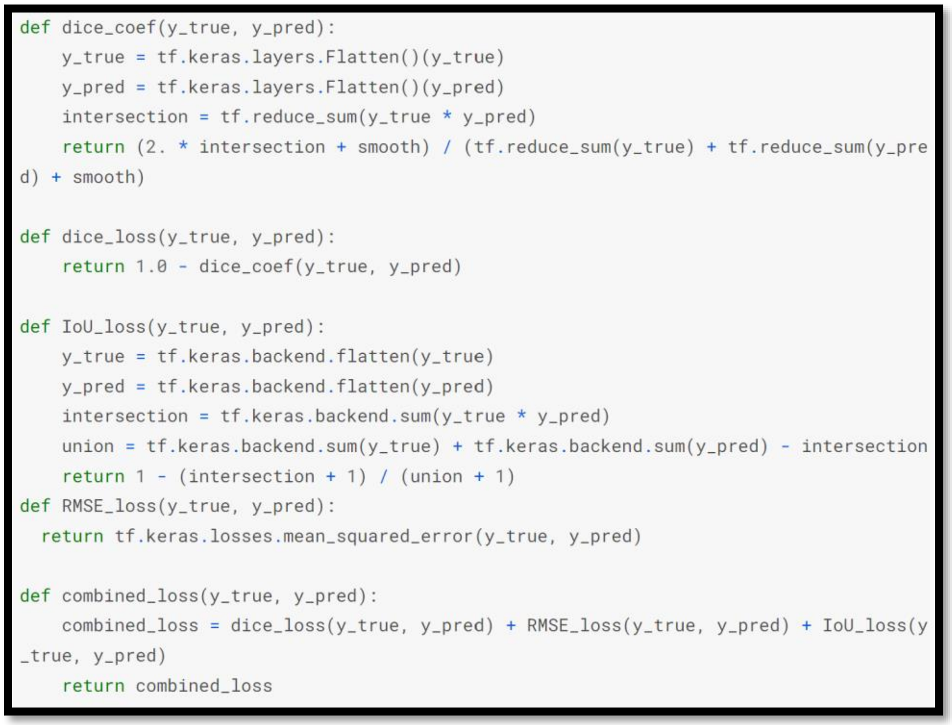
1- Upload a test image and its corresponding mask, and preformed preprocessing on them.



```
x = cv2.imread("/kaggle/input/tests-for-ensambling/test_R/31252.jpg", cv2.IMREAD_COLO
R)
x = cv2.cvtColor(x, cv2.COLOR_BGR2RGB)
x = cv2.resize(x, (256, 256))
test_R = x/255.0

IMAGE_SIZE = 256
y = cv2.imread("/kaggle/input/tests-for-ensambling/test_D/31252.png", cv2.IMREAD_GRAYS
CALE)
y = cv2.resize(y, (IMAGE_SIZE, IMAGE_SIZE))
y = np.expand_dims(y, axis=-1)
test_D = y/255.0
```

Code snippet 1.19: Uploading test image

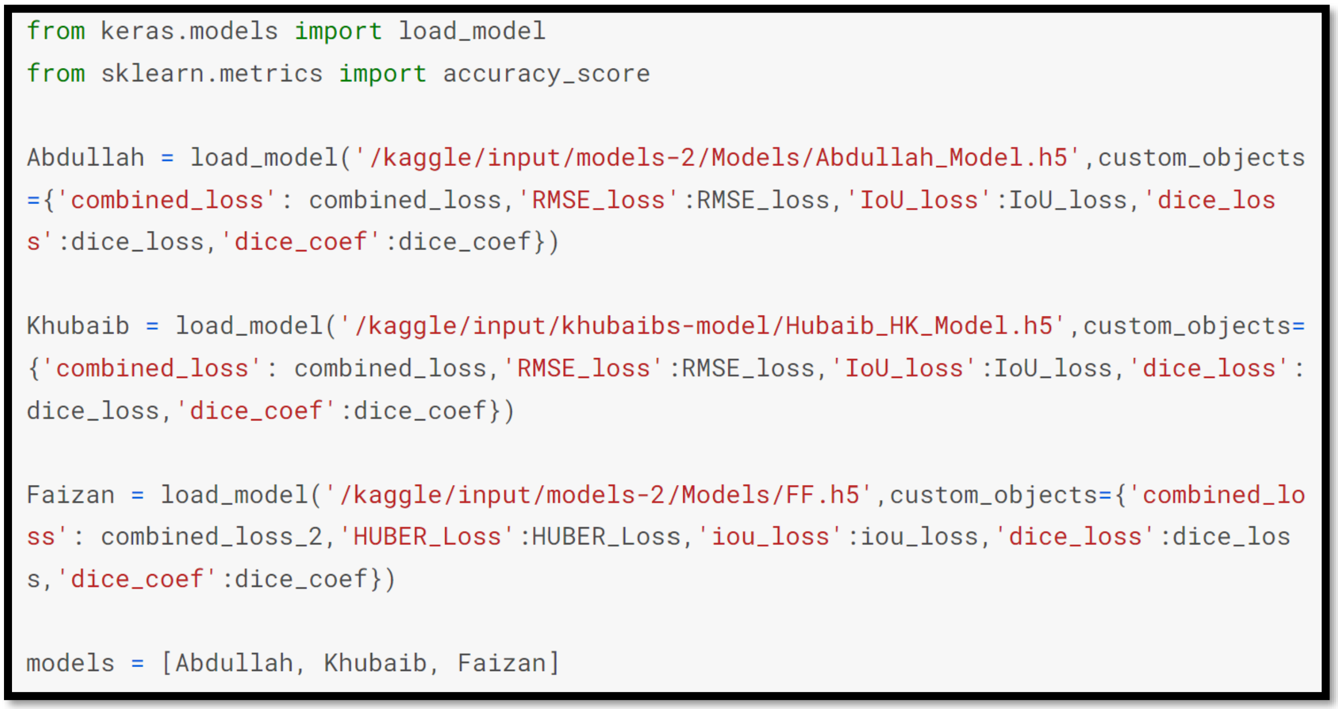
2- Re-define all the customized loss function used in each model, this step is necessary to load each model. lucky Abdullah and Khubaib used all the same functions , But In fizan's model we add one more Huber loss function.



```
def dice_coef(y_true, y_pred):
    y_true = tf.keras.layers.Flatten()(y_true)
    y_pred = tf.keras.layers.Flatten()(y_pred)
    intersection = tf.reduce_sum(y_true * y_pred)
    return (2. * intersection + smooth) / (tf.reduce_sum(y_true) + tf.reduce_sum(y_pre
d) + smooth)

def dice_loss(y_true, y_pred):
    return 1.0 - dice_coef(y_true, y_pred)

def IoU_loss(y_true, y_pred):
    y_true = tf.keras.backend.flatten(y_true)
    y_pred = tf.keras.backend.flatten(y_pred)
    intersection = tf.keras.backend.sum(y_true * y_pred)
    union = tf.keras.backend.sum(y_true) + tf.keras.backend.sum(y_pred) - intersection
    return 1 - (intersection + 1) / (union + 1)
def RMSE_loss(y_true, y_pred):
  return tf.keras.losses.mean_squared_error(y_true, y_pred)

def combined_loss(y_true, y_pred):
    combined_loss = dice_loss(y_true, y_pred) + RMSE_loss(y_true, y_pred) + IoU_loss(y
_true, y_pred)
    return combined_loss
```

Code snippet 1.20: redefining Loss Function.

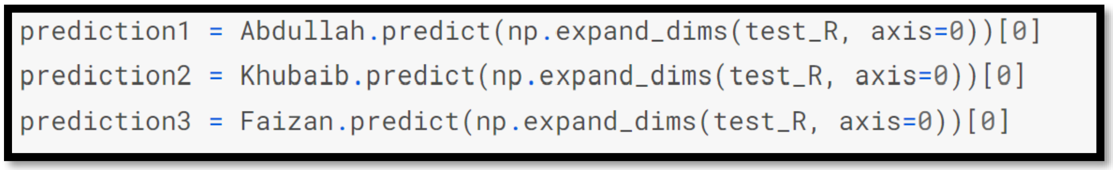
3- Load the model from the pre-saved (.h5) files of each model, including the customized loss function we used.



```
from keras.models import load_model
from sklearn.metrics import accuracy_score

Abdullah = load_model('/kaggle/input/models-2/Models/Abdullah_Model.h5',custom_objects
={'combined_loss': combined_loss,'RMSE_loss':RMSE_loss,'IoU_loss':IoU_loss,'dice_los
s':dice_loss,'dice_coef':dice_coef})

Khubaib = load_model('/kaggle/input/khubaibs-model/Hubaib_HK_Model.h5',custom_objects=
{'combined_loss': combined_loss,'RMSE_loss':RMSE_loss,'IoU_loss':IoU_loss,'dice_loss':
dice_loss,'dice_coef':dice_coef})

Faizan = load_model('/kaggle/input/models-2/Models/FF.h5',custom_objects={'combined_lo
ss': combined_loss_2,'HUBER_Loss':HUBER_Loss,'iou_loss':iou_loss,'dice_loss':dice_los
s,'dice_coef':dice_coef})

models = [Abdullah, Khubaib, Faizan]
```

Code snippet 1.21: loading the model with customized functions

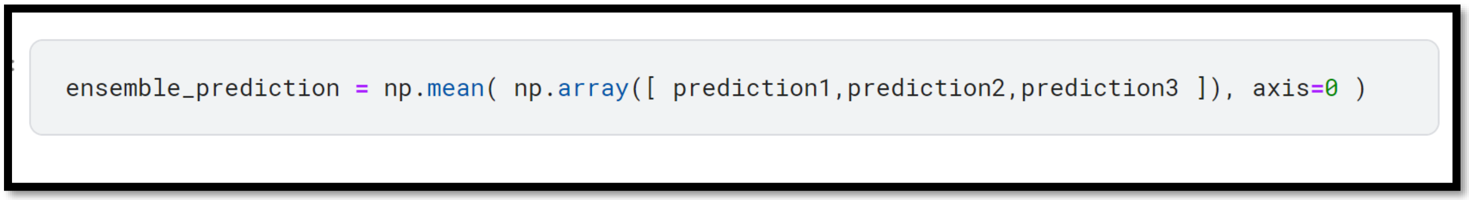
4- Feed the test image to each model to get those prediction & store the prediction in an array.



```
prediction1 = Abdullah.predict(np.expand_dims(test_R, axis=0))[0]
prediction2 = Khubaib.predict(np.expand_dims(test_R, axis=0))[0]
prediction3 = Faizan.predict(np.expand_dims(test_R, axis=0))[0]
```

Code snippet 1.22: Getting prediction of each model

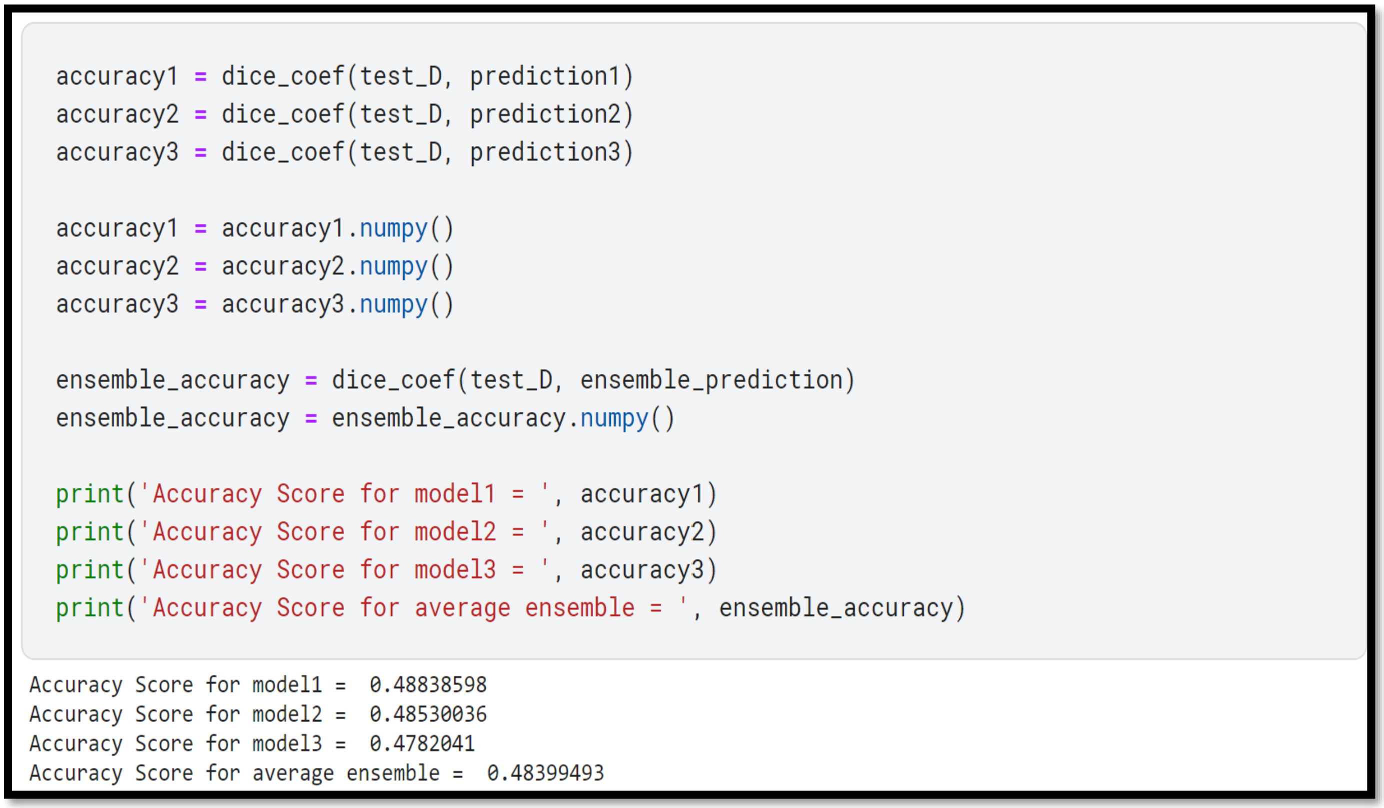
5- Ensemble using average:



```
ensemble_prediction = np.mean( np.array([ prediction1,prediction2,prediction3 ]), axis=0 )
```

Code snippet 1.23: ensemble all the prediction by taken mean

6- Calculate the Accuracy:



```
accuracy1 = dice_coef(test_D, prediction1)
accuracy2 = dice_coef(test_D, prediction2)
accuracy3 = dice_coef(test_D, prediction3)

accuracy1 = accuracy1.numpy()
accuracy2 = accuracy2.numpy()
accuracy3 = accuracy3.numpy()

ensemble_accuracy = dice_coef(test_D, ensemble_prediction)
ensemble_accuracy = ensemble_accuracy.numpy()

print('Accuracy Score for model1 = ', accuracy1)
print('Accuracy Score for model2 = ', accuracy2)
print('Accuracy Score for model3 = ', accuracy3)
print('Accuracy Score for average ensemble = ', ensemble_accuracy)
```

```
Accuracy Score for model1 =  0.48838598
Accuracy Score for model2 =  0.48530036
Accuracy Score for model3 =  0.4782041
Accuracy Score for average ensemble =  0.48399493
```

Code snippet 1.24: Calculate Accuracy

9-Final Results.

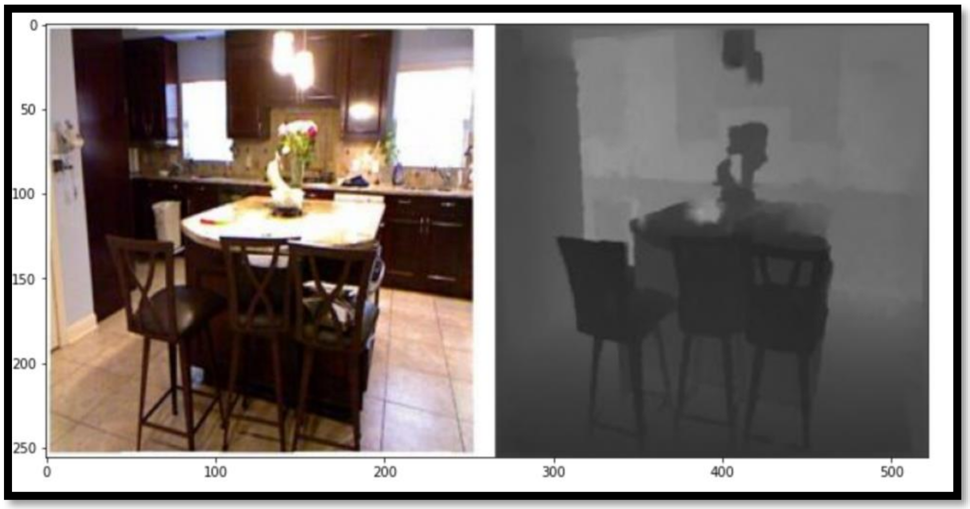


Figure 1.19: Input RGB and Corresponding true mask

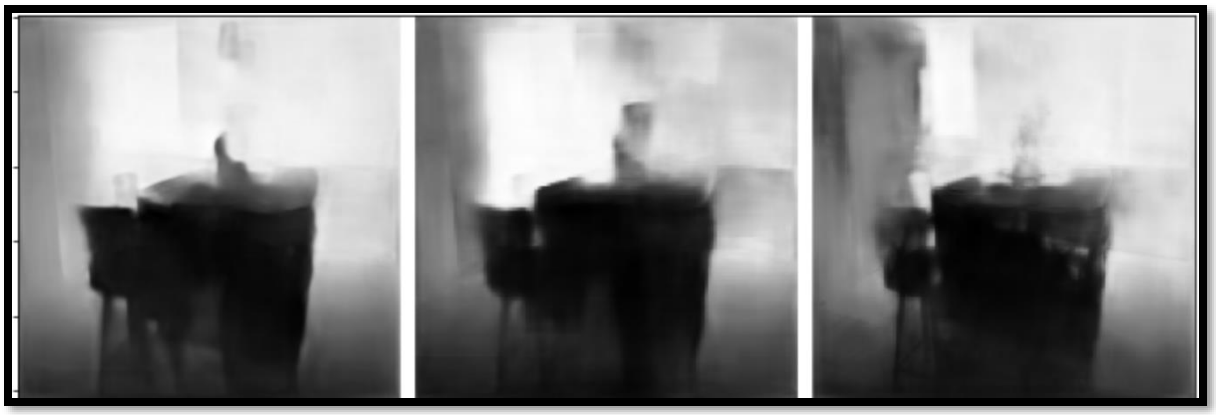


Figure 1.20: three predictions (Abdullah , Khubaib and Fizan respectively)

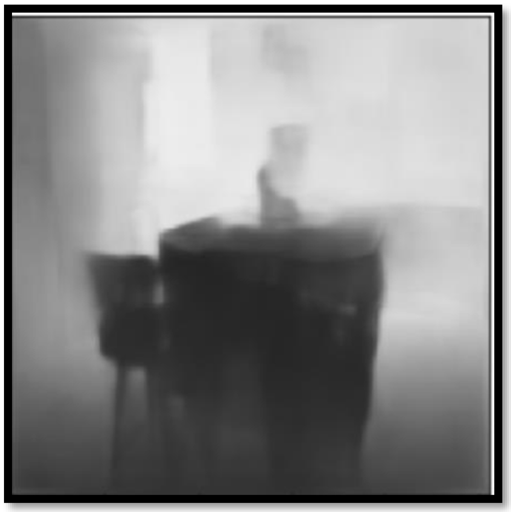


Figure 1.21: Ensembling Results

- Those images may not look clear on printing, please refer to the soft copy of the paper.

Conclusion:

The RGB-D depth estimation is a unique task, for it is NOT a binary segmentation between two classes , the RGB-D consider each pixel as a class and tries to segment pixel on basis of their distance away from the camera, because of this , the authors approach this segmentation as a regression segmentation somehow and ended up with a mask with continues value for each pixel.

And since that all three models that they build had very much close results, they decided to average out the three model and ended up with a reasonable solution.

For future work:

These models could be further enhanced if the authors had access to a strong processing power, for they may be able to increase there layers and more data, Anyhow the application of the RGB-D depth estimation is limitless, for example, this very same model can help a SLAM Robot navigate through the environment by using his camera to Capture RGB images and have this algorithm estimate depth for It.

Works Cited

Godard, Clément, Oisin Mac Aodha, Michael Firman, and Gabriel J. Brostow. n.d. "Godard, Clément, Oisin Mac Aodha, Michael Firman, and Gabriel J. Brostow. "Digging into self-supervised monocular depth estimation." In Proceedings of the IEEE/CVF International Conference on Computer Vision, pp. 3828-3838. 2019."

Ronneberger, O., Fischer, P., & Brox, T. (2015, October). U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention (pp. 234-241). Springer, Cham. n.d.

Code notebooks:

Model-1:https://www.kaggle.com/code/rafeytahir23/depth-segmentation (Look for Abdullah Abid version)

Model-2:https://www.kaggle.com/code/rafeytahir23/depth-segmentation (Look for Khubaib Hidar version)

Model-3:https://www.kaggle.com/code/abdullah7987/depth-segmentation(Look for Fizan Faiz version)

Ensembling & Utilization Code:

https://www.kaggle.com/code/abdullah362376/ensembling (Look for version_2)