HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Graduation Research 1

Graduation Research 1 Report

Extreme Lightweight Image Segmentation

Student	**Student ID**
Do Hoang Viet	20176908

Professor:
Ta Hai Tung

Graduation Research 1 Report

August 15, 2021

The report was written as part of Graduation Research 1 course.

Contents

1 Introduction

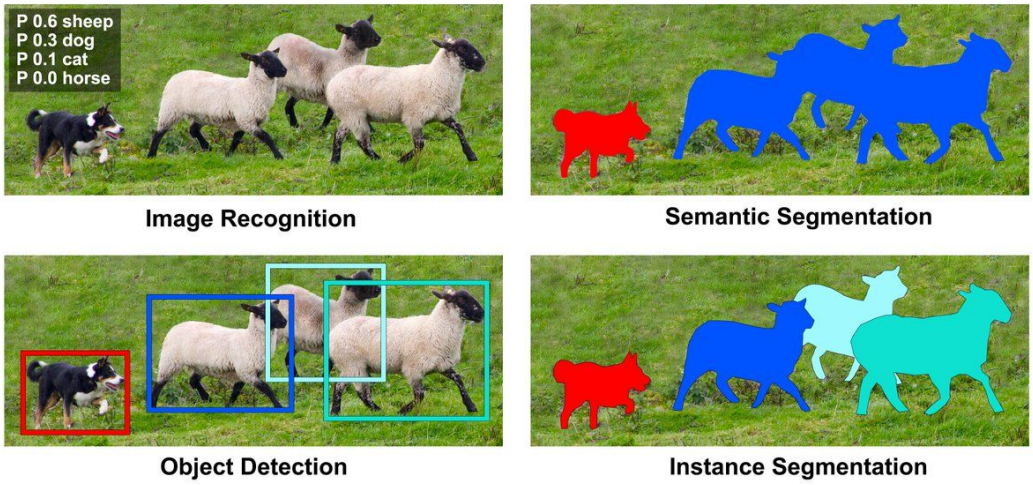

Figure 1: Image example for Image Recognition (top-left), Object Detection (bottom left) and Image segmentation

Image segmentation is one of main important problems in Machine Learning and Data Mining, beside Object Detection. While Object Detection finds the bounding box and class instance for each object in image, Image Segmentation finds class for each pixels in image. Image segmentation is divided into 2 different types:

- Instance Segmentation: Model is interested in different instances of object in the same object type
- Semantic Segmentation: Model is not interested in different instances of object in the same object type

2 Difficulty

Portrait segmentation has its unique requirements. First, because the portrait segmentation is performed in the middle of a whole process of many real-world applications, it requires extremely lightweight models. Second, there has not been any public datasets in this domain that contain a sufficient number of images with unbiased statistics. Here is these difficulty of image segmentation.

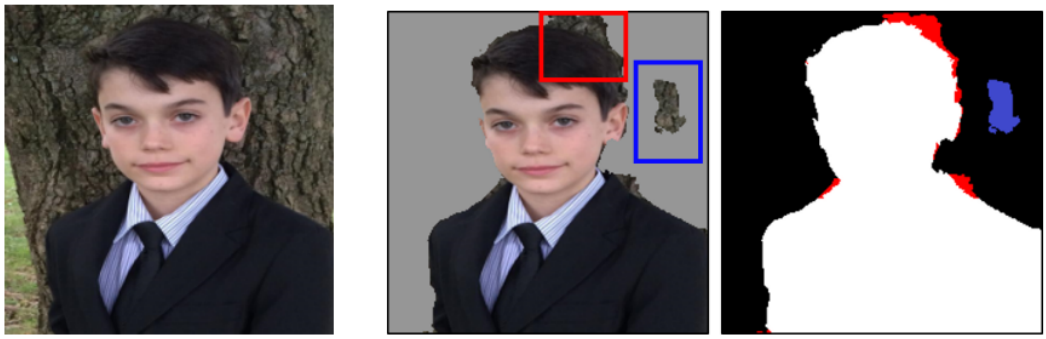

Figure 2: Difficulty of Image segmentation

2.1 Global Consistency

This is a common problem that segmentation models often encounter, the phenomenon that the model often incorrectly distinguishes between background features and object segment. The explanation for the above phenomenon is usually because the receptive fields in the model are not large enough to get the global context of the image =¿ model predict incorrectly. In the description for this, I can observe the blue spot in figure b above.

In order to increase global consistency the simplest way to think of is to use larger receptive fields, I can create Convs with larger kernel sizes, but this method has a disadvantage that can be easily obtained See: larger kernel equals more parameters, longer computational time leads to difficulty in practical application. For this same reason, the Dilated Conv concept came into being as an effective way to get larger fields and still retain localization information. However, as the dilation rate increases, the number of weights valid decreases.

Also, there is another way to create lightweight models with different receptive fields is to use Spatial Pyramid Pooling with other pool size and concatenate the features or multi-path structure for extraction features, but they all have one disadvantage. The point is high latency.

2.2 Preserving detailed local information

Another difficult issue facing segmentation models is how to recover and preserve detailed local information. This factor is extremely important to obtain sharpness at the edge of the segmentation mask or not. Observe the red spots in figure b above.

I understand that after the encoder process I get low resolution feature maps containing high level information and to obtain original resolution, I am interested in the concepts of Transposed Conv or Global Attention to Upsampling step by step. However, these attention vectors do not reflect well the local information caused by global pooling. Currently, a new method for achieving better results in segmentation is called two-branch method, typically **FastSCNN**, **ContextNet**, **BiSeNet**, a path for the global context, 1 for detailed information. The downside is that it still needs to be computed twice, each time for each branch.

3 Popular Image Segmentation Architecture

Key features of segmentation model is that the shape of output is the same as the shape of input. The main idea is based on high level features to predict the class for each pixel in image. These high level features contain important information such as edges, shape of objects, global context of image which is obtained from a Fully Convolutional Network (also known as Encoder). Based on these features, I need to reproduce spatial dimension by using Upsampling layers. This kind of architecture also known as **Encoder-Decoder**

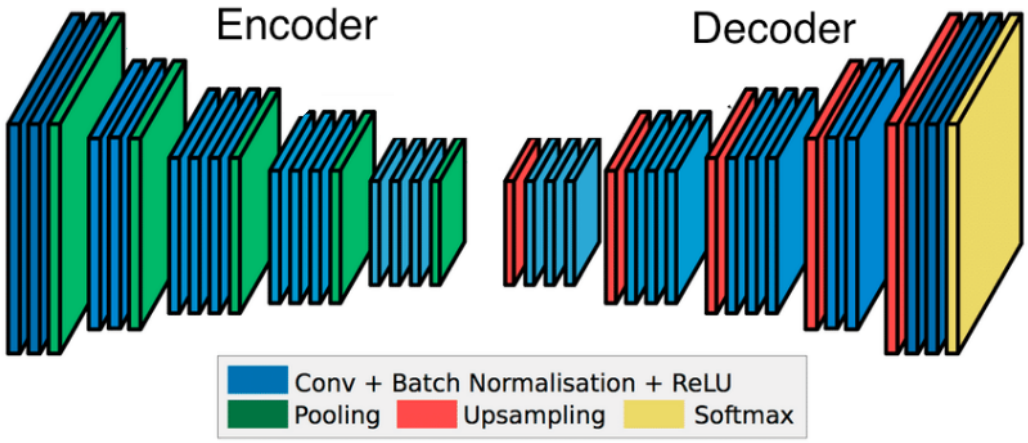

Figure 3: Encoder-Decoder architecture

The introduction of Skip Connection is quite important as it increases the accuracy of the output shapes, especially at boundaries positions. The reason is that when I upsampling from low-resolution tensors positions, some low-level information is lost in the encoder. To solve this problem, I allow each upsampling layer to be added with low-level features from the layers from its corresponding encoder. This increases the smoothness at the boundaries of the objects.

Here are some popular architectures:

- FCN: one of the first end-to-end semantic segmentation model, based on VGG or AlexNet.
- SegNet: follows encoder-decoder architecture.
- UNet: similar to SegNet, but utilize Skip Connection.
- PSPNet: optimized to solve global consistency problem, but leads to high latency problem.

In this project, I implemented UNet and SINet - a lightweight model - to separate human and background in an image and apply these model in a web application.

4 UNet

U-Net is a convolutional neural network that was developed for biomedical image segmentation at the Computer Science Department of the University of Freiburg, Germany. The network is

based on the fully convolutional network and its architecture was modified and extended to work with fewer training images and to yield more precise segmentations.

Some features of UNet:

- Works efficiently on small datasets through heavy data augmentation as in some cases the number of annotated samples will be less.
- Works for binary as well as multiple classes.
- It is scale-invariant.
- Can be used in a multitude of applications like Object Detection/Segmentation, GANs, etc.
- The features extracted at different levels in the contracting path are then combined with the feature maps in the expansion path giving rise to more number of features which is necessary for an accurate segmentation map.

4.1 Model

(Network design): img + overview (use encoder-decoder + skip connection) (Loss function)

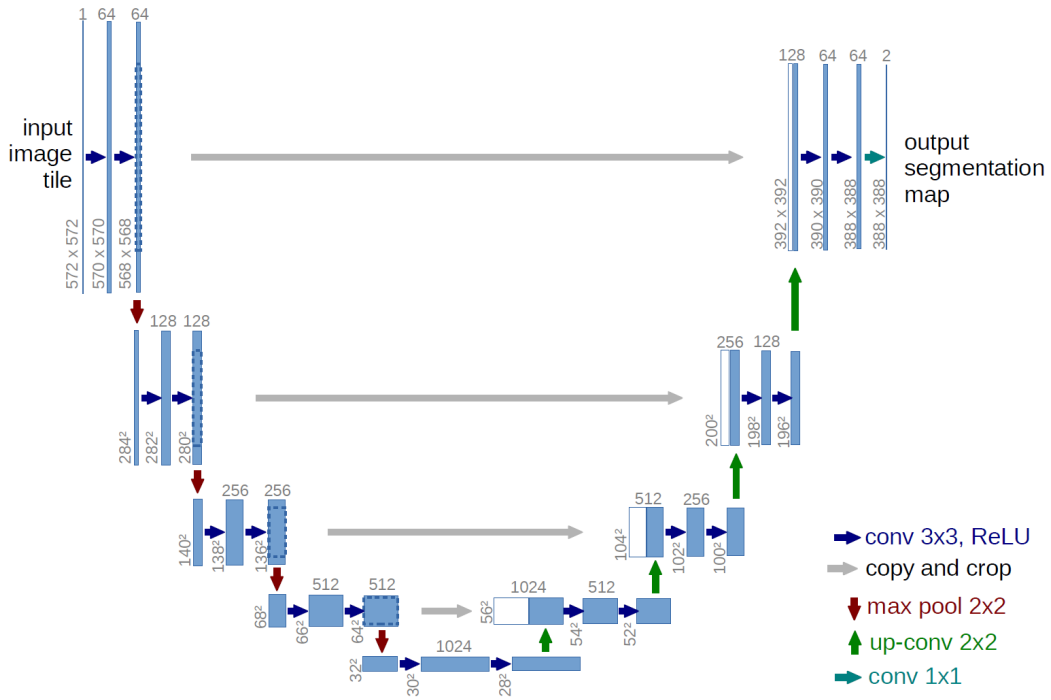

Figure 4: U-net architecture (example for 32x32 pixels in the lowest resolution)

The main difference between Unet and other semantic segmentation networks is to supplement a usual contracting network by successive layers, where pooling operations are replaced by upsampling operators. Hence these layers increase the resolution of the output. In order to

localize, high resolution features from the contracting path are combined with the upsampled output. A successive convolution layer can then learn to assemble a more precise output based on this information.

4.1.1 Network Architecture

The architecture looks like a 'U' which justifies its name. This network architecture consists of a contracting path (encoder - left side) and an expansive path (decoder - right side). The contracting path follows the typical architecture of a convolutional network. It consists of the repeated application of two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for downsampling. At each downsampling step I double the number of feature channels. Every step in the expansive path consists of an upsampling of the feature map followed by a 2x2 convolution ("up-convolution") that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes.

4.1.2 Skip Connections

Skip connections is a standard module in many convolutional architectures, it skips some layer in the neural network and feeds the output of one layer as the input to the next layers (instead of only the next one).
UNet architecture is long skip connections. As described from the figure, each step in the expansive path, after the transposed convolution, image is concatenated with the corresponding image from the contracting path and together makes an image with more precise prediction.

4.1.3 Loss Function

UNet uses a rather novel loss weighting scheme for each pixel such that there is a higher weight at the border of segmented objects. This loss weighting scheme helped the U-Net model segment cells in a discontinuous fashion such that individual cells may be easily identified within the binary segmentation map.

First of all pixel-wise softmax applied on the resultant image which is followed by cross-entropy loss function. So I am classifying each pixel into one of the classes. The idea is that even in segmentation every pixel have to lie in some category and I just need to make sure that they do. So I just converted a segmentation problem into a multiclass classification one and it performed very well as compared to the traditional loss functions.

4.2 Training

4.2.1 General

The model is implemented using Tensorflow Keras and can be trained on Google Colaboratory using GPU. However, to save time, I used pretrained model instead.

4.2.2 Dataset

The dataset consists of 18698 human portrait images of size 128x128 in RGB format, along with their masks(ALPHA), both are stored in Numpy array format.

```
[ ]
    # Load the dataset
    x_train=np.load("/content/data/img_uint8.npy")
    y_train=np.load("/content/data/msk_uint8.npy")
```

```
[ ]  # Verify the mask shape and values
    print(np.unique(y_train))
    print(y_train.shape)

    # Total number of images
    num_images=x_train.shape[0]
```

```
    [  0 255]
    (18698, 128, 128, 1)
```

Figure 5: Implemented code for verifying dataset

4.2.3 Datagenerator

To increase the size of dataset and model robustness, cropping, brightness alteration, flipping, resize image, increase contrast, mean-substracttion etc.. are utilized. Since most of the images contain plain background, new synthetic images are created using random backgrounds (natural) using the default dataset, with the help of Keras.
A data generator is used to load images and masks together at runtime and use the same seed for performing run-time augmentation for images and masks using 80/20 tran-val split.

```
# Data generator for training and validation

data_gen_args = dict(rescale=1./255,
                     width_shift_range=0.1,
                     height_shift_range=0.1,
                     zoom_range=0.2,
                     horizontal_flip=True,
                     validation_split=0.2
                    )

image_datagen = ImageDataGenerator(**data_gen_args, preprocessing_function=normalize_batch)
mask_datagen = ImageDataGenerator(**data_gen_args,  preprocessing_function=normalize_batch)

# Provide the same seed and keyword arguments to the fit and flow methods
seed = 1
batch_sz=BATCH_SIZE

# Train-val split (80-20)
num_train=int(num_images*0.8)
num_val=int(num_images*0.2)
```

Figure 6: Datagenerator

4.2.4 Normalize

Normalize the source images at runtime; but do not modify the masks

```
[ ]  # Preprocessing function (runtime)
     def normalize_batch(imgs):
         if imgs.shape[-1] > 1 :
           return (imgs -  np.array([0.50693673, 0.47721124, 0.44640532])) /np.array([0.28926975, 0.27801928, 0.28596011])
         else:
           return imgs.round()
     def denormalize_batch(imgs,should_clip=True):
         imgs= (imgs * np.array([0.28926975, 0.27801928, 0.28596011])) + np.array([0.50693673, 0.47721124, 0.44640532])

         if should_clip:
             imgs= np.clip(imgs,0,1)
         return imgs
```

Figure 7: Implemented code for normalizing function

4.2.5 Data Augmentation

The model use Mobilent v2 with depth multiplier 0.5 as encoder (feature extractor).
For the decoder part, it used resize bilinear for upsampling, along with Conv2d. Skip connections are applied between encoder and decoder parts for better results.

4.2.6 Training

Model is trained with 300 epochs and Adam optimizer, learning rate is 1e-3, activate function is sigmoid. For loss calculation, binary crossentropy is used

```
# Model architecture
def get_mobile_unet(finetune=False, pretrained=False):

    # Load pretrained model (if any)
    if (pretrained):
       model=load_model(PRETRAINED)
       print("Loaded pretrained model ...\n")
       return model

    # Encoder/Feature extractor
    mnv2=keras.applications.mobilenet_v2.MobileNetV2(input_shape=(128, 128, 3),alpha=0.5, include_top=False, weights='imagene
t')

    if (finetune):
      for layer in mnv2.layers[:-3]:
        layer.trainable = False


    x = mnv2.layers[-4].output

    # Decoder
    x = deconv_block(x, 512)
    x = concatenate([x, mnv2.get_layer('block_13_expand_relu').output], axis = 3)

    x = deconv_block(x, 256)
    x = concatenate([x, mnv2.get_layer('block_6_expand_relu').output], axis = 3)

    x = deconv_block(x, 128)
    x = concatenate([x, mnv2.get_layer('block_3_expand_relu').output], axis = 3)

    x = deconv_block(x, 64)
    x = concatenate([x, mnv2.get_layer('block_1_expand_relu').output], axis = 3)


    x = Conv2DTranspose(filters=32, kernel_size=3, strides=2, padding='same', kernel_initializer = 'he_normal')(x)
    #x = UpSampling2D(size = (2,2),interpolation='bilinear')(x)
    #x = Conv2D(filters=32, kernel_size=3, padding = 'same', kernel_initializer = 'he_normal')(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)


    x = Conv2DTranspose(1, (1,1), padding='same')(x)
    x = Activation('sigmoid', name="op")(x)


    model = Model(inputs=mnv2.input, outputs=x)


    model.compile(loss='binary_crossentropy', optimizer=Adam(lr=1e-3),metrics=['accuracy'])
    return model
```

Figure 8: Implemented code for unet model

```
# Train the model
model.fit_generator(
    train_generator,
    epochs=300,
    steps_per_epoch=num_train/batch_sz,
    validation_data=val_generator,
    validation_steps=num_val/batch_sz,
    use_multiprocessing=True,
    workers=2,
    callbacks=callbacks_list)
```

Figure 9: Implemented code for training step

After 300 epochs, loss: 0.0588 - accuracy: 0.9656

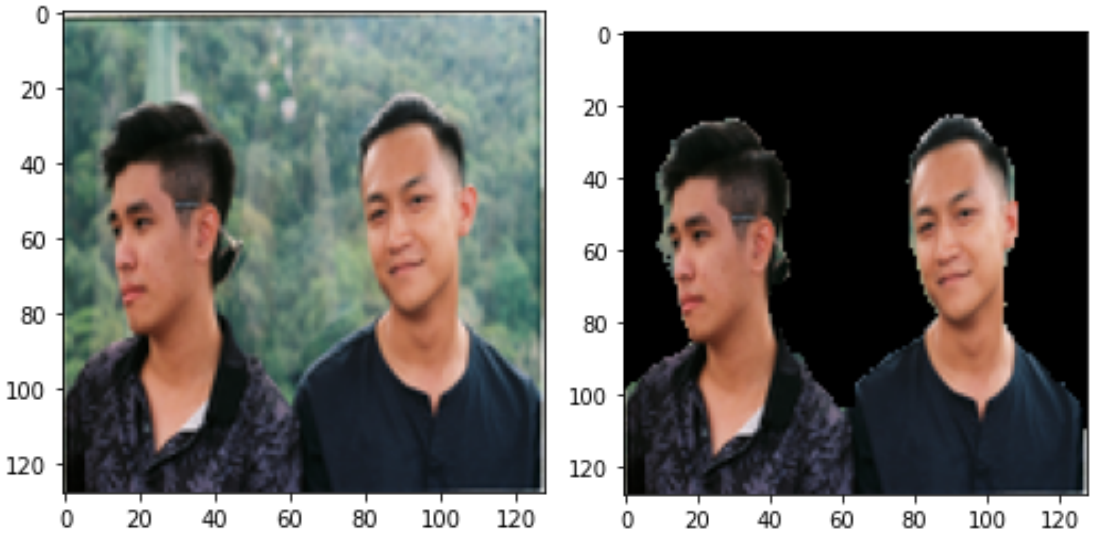

Figure 10: Result of prediction

4.2.7 Evaluation and Optimization

Evaluation of the performance of the model on the test dataset.

```
[ ]  # Evaluate model
     score = model.evaluate(np.float32(new_xtest/255.0), np.float32(new_ytest/255.0), verbose=0)
     # Print loss and accuracy
     print('Test loss:', score[0])
     print('Test accuracy:', score[1])

     Test loss: 0.2071368247270584
     Test accuracy: 0.9180577993392944
```

Figure 11: Evaluation

Using the kito library, the model can be optimized by folding the batch norms. This does not change the model behaviour or accuracy, but helps to reduce the number of layers.

```
[ ]  # Optimize model by folding batch-norms
     model_reduced = reduce_keras_model(model)
     model_reduced.save('bilinear_bnoptimized_munet.h5')
```

Figure 12: Optimization

5 SINet

In Machine Learning and Data Mining, specifically in Image Segmentation problems, lightweight models capable of running in realtime, embedded into mobile devices while ensuring accuracy

have become a research topic in recent years. For that reason, the new extremely lightweight portrait segmentation model **SINet** - top 1 ranking on the EG1800 dataset according to Paperswithcode - were introduced, containing an information blocking decoder and spatial squeeze modules. The information blocking decoder uses confidence estimates to recover local spatial information without spoiling global consistency. The spatial squeeze module uses multiple receptive fields to cope with various sizes of consistency in the image. For the second problems, I use a datassets named **Nukki/baiduV1**, this dataset contains images with mask is obtained from a face recognition model.

5.1 Model

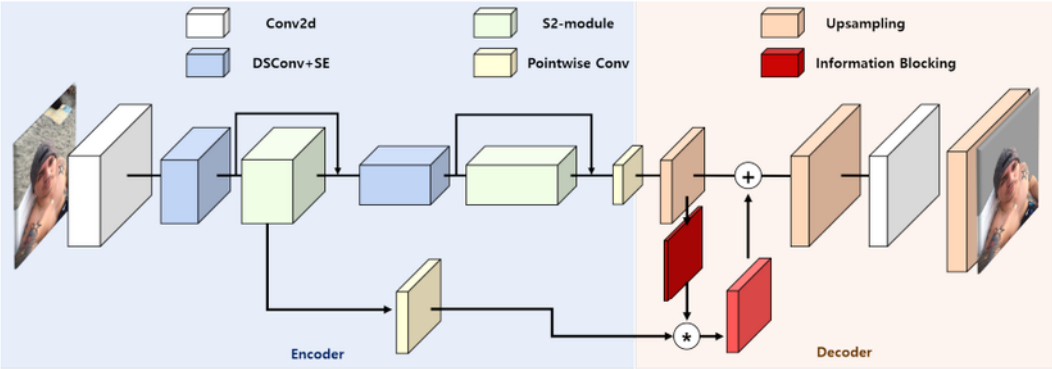

Figure 13: The overall architecture of SINet

The improvement of the SINet model can be summarized as follows:

- Introducing Information Blocking Decoder which helps to select low-level information for Upsampling layers instead of everything like traditional models.
- The appearance of Spatial Squeeze Module (S2-module) solved the problem of global consistency by utilizing multi-receptive field without sacrificing latency.
- Improvement in loss function, which solve mask boundaries problem.

5.1.1 Information Blocking Decoder

The information blocking decoder is designed to only take the necessary information from the high-resolution feature maps by utilizing the confidence score of the low-resolution feature maps while blocking the information flow from the high-resolution feature into the region where the encoder successfully segmented with high confidence. This prevents noisy information to ruin already certain areas, and allows the model to focus on regions with high uncertainty.

First, the model projects the last set of feature maps of the encoder to the size of the number of classes by a pointwise convolution and uses a bilinear upsampling to make the same resolution as the high-resolution target segmentation map. Then, the model employs a softmax function to get a probability of each class and calculates each pixel's confidence score C by taking maximum value among the probabilities of each class. Finally, I generate an information blocking map by computing (1 - C). I perform pointwise multiplication between the information blocking map and

the high-resolution feature maps. This ensures that low confidence regions get more information from the high-resolution feature maps, while high confidence regions keep their original values in the subsequent pointwise addition operation.

5.1.2 Spatial Squeeze module

Spatial Squeeze module (S2-module) handles global consistency by using the multireceptive field scheme, and squeezes the feature resolution to mitigate the high latency of multi-path structures.

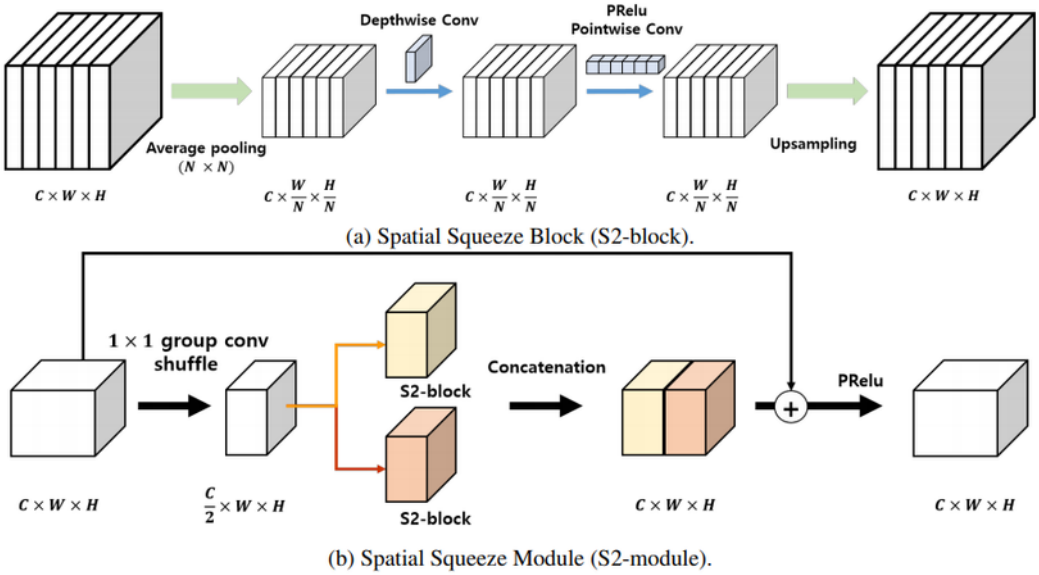

Figure 14: Spatial Squeeze Module (S2-module)

The main idea is to following split-transform-merge mechanism:

- First, a pointwise convolution is used to reduce the number of feature maps by half.
- The reduced feature maps then pass through each S2-block.
- The results are merged through concatenation.
- A residual connection is applied between the input feature map and the merged feature map.
- Finally, PRelu is utilized for non-linearity

For S2-block, average pooling is applied instead of dilated convolution for adjusting the size of the receptive field and reducing the latency.

5.1.3 Loss Function

Beside Cross Entropy Loss, SINet uses an additional Boundary Loss to improve accuracy at the boundaries of the segmented mask. The final loss is as follows:

$$\mathcal{B} = (f \oplus y^*) - (f \ominus y^*)$$
$$Loss = CE_{i \in \mathcal{P}}(y_i^*, \hat{y}) + \lambda CE_{j \in \mathcal{B}}(y_j^*, \hat{y}_j).$$

Figure 15: SINet final loss

Here, y* is a binary ground truth, yˆ is a predicted label from a segmentation model, λ is a hyperparameter that controls the balance between the loss terms. f is a (15 × 15) filter used for the morphological dilation and erosion operations on ground truth. P denotes all the pixels of the ground truth, and B denotes the pixels in the boundary area as defined by the morphology operation.

5.1.4 Network Design

	Input	Operation	Output	
1	3 × 224 × 224	CBR	12 × 112 × 112	Down sampling
2	12 × 112 × 112	DSConv+SE	16 × 56 × 56	Down sampling
3	16 × 56 × 56	SB module	48 × 56 × 56	[k=3, p=1], [k=5, p=1]
4	48 × 56 × 56	SB module	48 × 56 × 56	[k=3, p=1], [k=3, p=1]
5	64 × 56 × 56	DSConv+SE	48 × 28 × 28	Concat [2, 4], Down sampling
6	48 × 28 × 28	SB module	96 × 28 × 28	[k=3, p=1], [k=5, p=1]
7	96 × 28 × 28	SB module	96 × 28 × 28	[k=3, p=1], [k=3, p=1]
8	96 × 28 × 28	SB module	96 × 28 × 28	[k=5, p=1], [k=3, p=2]
9	96 × 28 × 28	SB module	96 × 28 × 28	[k=5, p=2], [k=3, p=4]
10	96 × 28 × 28	SB module	96 × 28 × 28	[k=3, p=1], [k=3, p=1]
11	96 × 28 × 28	SB module	96 × 28 × 28	[k=5, p=1], [k=5, p=1]
12	96 × 28 × 28	SB module	96 × 28 × 28	[k=3, p=2], [k=3, p=4]
13	96 × 28 × 28	SB module	96 × 28 × 28	[k=3, p=1], [k=5, p=2]
14	144 × 28 × 28	1x1 conv	$\#class$ × 28 × 28	Concat [5, 13]

Figure 16: Detailed settings for the SINet encoder. k denotes the kernel size of the depthwise convolution and p denotes the kernel size of average pooling the S2-block.

Some notable things about SINet model:

- SINet uses S2-modules as bottlenecks for reducing the resolution of feature maps.
- After each bottleneck S2-module, a residual connection is applied between bottleneck's input and its output, which help the module train longer

5.2 Training

5.2.1 General

The Sinet model is implemented using Tensorflow Keras, it is trained on Google Colaboratory using GPU. The dataset is loaded from Google Drive. The total training time is about 10 hours.

5.2.2 Dataset

Sinet model was trained and evaluated by the authors on EG1800 dataset. The EG1800 dataset, an accessible public portrait segmentation dataset, contains only around 1,300 training images, and has large biases with regard to attributes such as race, age, and gender. Thus, the authors have generated data from Baidu dataset which contains 5382 human full body segmentation images covering various poses, fashions and backgrounds. The dataset is generated using a Face detector, then they expanded the predicted region to get the portrait and segment for this portrait. They also crawled more data and used a pretrained segment model **DeepLabv3 + SE-ResNeXt-50** to obtain more data.

5.2.3 Datagenerator

Before training model, I need to a have a Datagenerator class which help train model by batch. I also need to do some preprocess steps such as: resize image, increase contrast, mean-substracttion, data augmentation.

5.2.4 Normalize

Mean substraction ([103.94, 116.78, 123.68], RGB) with imageval (0.017) was applied

```
190    def mean_substraction(self, image, mean=[103.94, 116.78, 123.68], image_val=0.017):
191        image = image.astype("float32")
192        image[:, :, 0] -= mean[0]
193        image[:, :, 1] -= mean[1]
194        image[:, :, 2] -= mean[2]
195        image *= image_val
196        image = image[..., ::-1]
197
198        return image
```

Figure 17: Implemented code for mean substraction

5.2.5 Data Augmentation

The following methods augmentations was applied before training are:

1. Geometric augmentation:
    - random horizontal flip
    - random rotation -45 degree to 45 degree
    - random resizing 0.5 to 1.5
    - random translation -0.25 to 0.25

```python
def _load_augmentation_aug_geometric(self):
    return iaa.Sequential([
        iaa.Sometimes(0.5, iaa.Fliplr()),
        iaa.Sometimes(0.5, iaa.Rotate((-45, 45))),
        iaa.Sometimes(0.5, iaa.Affine(
            scale={"x": (0.5, 1.5), "y": (0.5, 1.5)},
            order=[0, 1],
            mode='constant',
            cval=(0, 255),
        )),
        iaa.Sometimes(0.5, iaa.Affine(
            translate_percent={"x": (-0.25, 0.25), "y": (-0.25, 0.25)},
            order=[0, 1],
            mode='constant',
            cval=(0, 255),
        )),
    ])
```

Figure 18: Implemented code for geometric augmentation

2. Non-geometric augmentation:

   - random noise Gaussian noise, sigma = 10
   - image blur kernel size is 3 and 5 randomly
   - random color change 0.4 to 1.7
   - random brightness change 0.4 to 1.7
   - random contrast change 0.6 to 1.5
   - random sharpness change 0.8 to 1.3

```python
def _load_augmentation_aug_non_geometric(self):
    return iaa.Sequential([
        iaa.Sometimes(0.5, iaa.AdditiveGaussianNoise(loc=0, scale=(0.0, 0.05*255), per_channel=0.5)),
        iaa.Sometimes(0.5, iaa.OneOf([
            iaa.GaussianBlur(sigma=(0.0, 3.0)),
            iaa.GaussianBlur(sigma=(0.0, 5.0))
        ])),
        iaa.Sometimes(0.5, iaa.MultiplyAndAddToBrightness(mul=(0.4, 1.7))),
        iaa.Sometimes(0.5, iaa.GammaContrast((0.4, 1.7))),
        iaa.Sometimes(0.5, iaa.Multiply((0.4, 1.7), per_channel=0.5)),
        iaa.Sometimes(0.5, iaa.MultiplyHue((0.4, 1.7))),
        iaa.Sometimes(0.5, iaa.MultiplyHueAndSaturation((0.4, 1.7), per_channel=True)),
        iaa.Sometimes(0.5, iaa.LinearContrast((0.4, 1.7), per_channel=0.5))
    ])
```

Figure 19: Implemented code non geometric augmentation

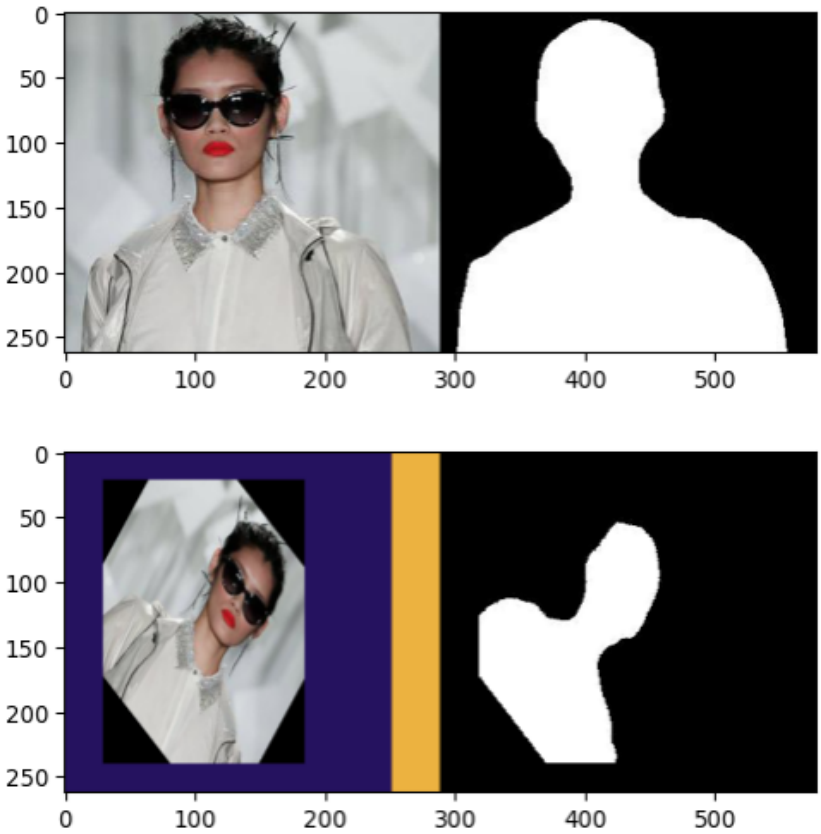

Figure 20: Result of augmentation

5.2.6 Loss Function

For training process, I build a custom loss class, which includes 1 function for calculating **Cross Entropy** and 1 function for calculating **Boundary Loss**

```python
class SINetLoss:
    def __init__(self, lamda=0.9):
        self.lamda = lamda

    def gt_dilation(self, y_true):
        y_true = tf.reshape(y_true, (-1, IMG_HEIGHT, IMG_WIDTH, N_CLASSES))
        dilation = tf.nn.max_pool2d(y_true, ksize=(15, 15), strides=1, name='dilation2D', padding="SAME")
        dilation = tf.reshape(dilation, (-1, IMG_HEIGHT*IMG_WIDTH, N_CLASSES))
        return dilation

    def gt_erosion(self, y_true):
        y_true = tf.reshape(y_true, (-1, IMG_HEIGHT, IMG_WIDTH, N_CLASSES))
        erosion = -tf.nn.max_pool2d(-y_true, ksize=(15, 15), strides=1, name='erosion2D', padding="SAME")
        erosion = tf.reshape(erosion, (-1, IMG_HEIGHT*IMG_WIDTH, N_CLASSES))
        return erosion

    def log_loss(self, y_true, y_pred):
        loss = -tf.reduce_sum(y_true * tf.math.log(y_pred), axis=-1)
        return loss

    def boundary_loss(self, y_true, y_pred):
        dilation = self.gt_dilation(y_true)
        erosion = self.gt_erosion(y_true)
        boundary = tf.math.subtract(dilation, erosion)
        loss = -tf.reduce_sum(boundary * tf.math.log(y_pred), axis=-1)
        return loss

    def compute_loss(self, y_true, y_pred):
        batch_size = tf.shape(y_pred)[0]
        self.lamda = self.lamda
        log_loss_ = tf.cast(self.log_loss(y_true, y_pred), tf.float32)
        boundary_loss_ = tf.cast(self.boundary_loss(y_true, y_pred), tf.float32)
        total_loss = log_loss_ + self.lamda * boundary_loss_
#         total_loss = tf.cast(total_loss, tf.float32)
        total_loss *= tf.cast(batch_size, tf.float32)

        return total_loss
```

Figure 21: Implemented code for loss function

1. **Cross entropy** is calculated for all the pixels of output result
2. **Boundary loss** is calculated by using 2 operations **dilation** and **erosion morphology** with window size = (15x15) on ground truth. Boundary loss is obtained by substraction element-wise between 2 matrices dilation and erosion morphology. **Boundary loss** will be the value of cross entropy between ground truth and predicted label on the pixels in boundary.
3. In the last part, I multiply between loss with batch size, that helped the model converge faster.

5.2.7 Training

Model is trained with 600 epochs and Optimizer is Adam, initial learning rate is 7.5e-3 with weight decay = 2e-4. first 300 epochs, I only trained encoder with batch size=36, last 300 epochs I opened all the layers and trained with batch size=24. I also apply LearningRateScheduler callback.

```
sinet_loss = SINetLoss().compute_loss
```

```
init_lr = 7.5e-3

opt = Adam(lr=init_lr, decay=2e-4, beta_1=0.9, beta_2=0.999, epsilon=1e-07,)
```

```
model.compile(optimizer=opt, loss=sinet_loss, metrics=["accuracy"])
```

```
trained_weight = "trained_weight"

if not os.path.exists(trained_weight):
    os.mkdir(trained_weight)
```

```
import keras.backend as K
import keras.callbacks as cbks
```

```
class CustomMetrics(cbks.Callback):
    def on_epoch_end(self, epoch, logs=None):
        for k in logs:
            if k.endswith('boundary_loss'):
                print(logs[k])
```

```
checkpoints = ModelCheckpoint(filepath=os.path.join(trained_weight, "best_weights_4_all.h5"),
                              monitor="val_loss",
                              save_weights_only=True,
                              save_best_only=True,
                              verbose=1)
```

Figure 22: Implemented code for training step

Open/Close layers

```
for idx, layer in enumerate(model.layers):
    print(idx, layer.name, layer.trainable)

    layer.trainable = True
    if idx > 361:
        layer.trainable = False
```

Figure 23: Implemented code for opening and closing layer

Poly Decay:

```python
def poly_decay(epoch):
    maxEpochs = 600
    baseLR = INIT_LR
    power = 1.0
    alpha = baseLR * (1 - (epoch / float(maxEpochs))) ** power
    return alpha
```

Figure 24: Implemented code for poly decay

Training:

```python
H = model.fit_generator(train_datagen.generate(),
                        epochs=600,
                        steps_per_epoch=train_datagen.get_n_examples() // train_datagen.batch_size,
                        validation_data=val_datagen.generate(),
                        validation_steps=val_datagen.get_n_examples() // val_datagen.batch_size,
                        callbacks=[checkpoints, LearningRateScheduler(poly_decay)],
                        initial_epoch=319
                        )
```

Figure 25: Implemented code training step

After 600 epochs, I obtained **loss: 8.3709 - accuracy: 0.9729 - val loss: 10.2774 - val accuracy: 0.9573**.

```python
predict()
```

Figure 26: Result of prediction

6 Application

I apply UNet and SINet in a segmentation problem - remove background from a human portrait. It is a web application which user can upload an image, select a segmentation model and background color. After a few seconds, a new image will be generated with its original background replaced by the selected color. This application is suitable for creating ID card's image or simply used to test the accuracy of my segmentation models.

6.1 Server

The server is built using Flask - a lightweight web application framework - with Python as its main programming language. It is easy to setup and deploy, since no database is required as the uploaded and generated images will be stored directly on the server. In addition, the server is also integrated with Keras to load model and perform prediction using trained weights.

The server works as follows:

- Preload models and weights on startup.
- Render the HTML page as user enter the website.
- Receive request including image file, model type and background color in HEX format.
- Resize image and put it into selected model to predict
- Get the image's mask from prediction and use it to replace background with selected color.
- Both original and generated image are stored on the server.
- Display the results page with original and generated image for comparison.

6.2 Client

The client is built on HTML, CSS with jQuery, generating two separated screen: Upload screen and Result screen
Upload screen has all the instructions that users need to modify background color and upload image

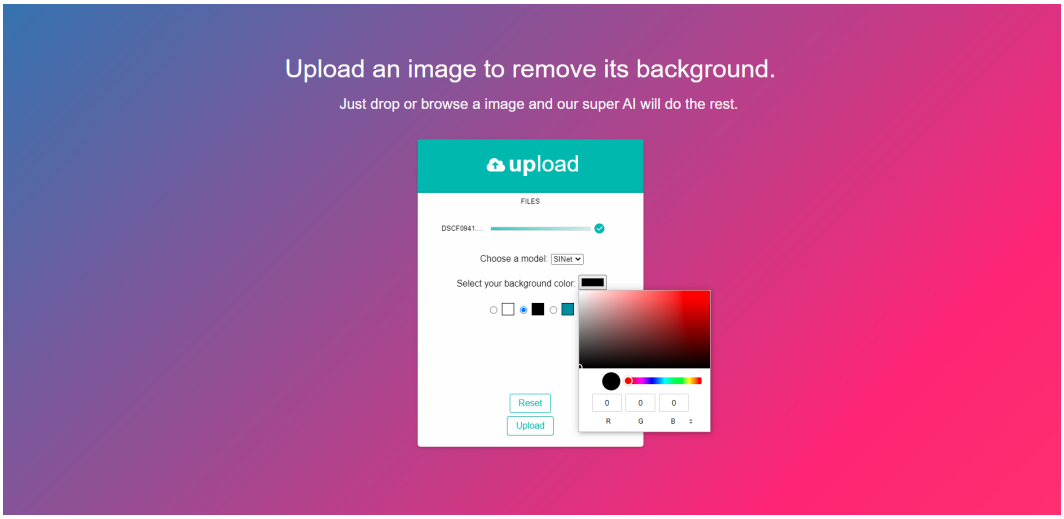

Figure 27: Upload screen

Steps to use my Image Segmentation:

- The first screen require users to drop an image or import it from user's device.
- After image is loaded succesfully, models and background color is optional with default model is SINet and default background color is black.
- Upload button would redirect users to **Result screen**.
- Reset button would allow user to remove loaded image and be able to upload another image.

Result screen show the result to users, which divides screen into two half for images before and after segmentation. Also users have option to return to upload screen.

7 Conclusion

Image segmentation in real life has many applications such as Content-based image retrieval, Medical imaging, Traffic control systems, etc... After this project, I have successfully tried and applied 2 portrait segmentation models. The Unet model has some traditional problems that I have mentioned. Thus, I have implemented Sinet in Keras tensorflow and applied it to real world application (remove background from portrait image).

Result evaluation: I have trained SINet with accuracy 95% and UNet with accuracy 94%. However, the result image of UNet is not good as SINet (Global Consistency problem). Beside the accuracy, the resolution of result image is low compared to original image. The reason for this problem is that I have to resize to image to a fixed size (128x128 for UNet, 224x224 for SINet) to achieve light weight prediction step. The accuracy is not good as in traditional image segmentation model, but I achieve light weight model which can be embedded in mobile device.

Improvement in future:

- To improve accuracy of model, I may try to use pretrained model and continue to train with more datas, more epochs in the current checkpoint
- For the resolution problem, I may need to custom input layer of the model to train with larger size of image that may lead to computational cost, slower prediction speed
- I may try to embed the model in a mobile application to see the application (needed for many cases in real life)
- Currently, my application only load the weight from file. I may try to implement pipeline workflow includes preprocess, training, predict as a life cycle that can be a service needed by other companies.

Resources:

- Github link: https://github.com/hvietdo99/GR1-20176908
- Google Colab notebook:
  - **SINet:** `https://colab.research.google.com/drive/1oeDTSAEGunwFH0_P4VTaZ8KJpWG4-adY?authuser=1#scrollTo=5eoyLlqN0XES`
  - **UNet:** `https://colab.research.google.com/drive/1-ozPj3Kap75Jg_3pTy4N8tNZyYWsSJbi?authuser=1#scrollTo=9gFX_Wltf-7f`

References

- https://arxiv.org/pdf/1911.09099.pdf
- https://github.com/titu1994/keras-squeeze-excite-network
- https://arxiv.org/pdf/1709.01507.pdf
- https://www.yongliangyang.net/docs/mobilePotrait_c&g19.pdf
- https://github.com/aleju/imgaug
- https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-ker
  html
- https://wacv20.wacv.net/
- https://paperswithcode.com/task/portrait-segmentation
- https://keras.io/
- https://arxiv.org/abs/1803.06815
- https://arxiv.org/abs/1505.04597
- https://towardsdatascience.com/unet-line-by-line-explanation-9b191c76baf5
- https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47
- https://github.com/anilsathyan7/Portrait-Segmentation
- https://codepen.io/P1N2O/pen/pyBNzX
- https://codepen.io/jotavejv/pen/bRdaVJ
- https://flask.palletsprojects.com/en/1.1.x/