Real-Time High-Resolution Background Matting

Samuel Golden, A20430084
Theo Guidroz, A20426895

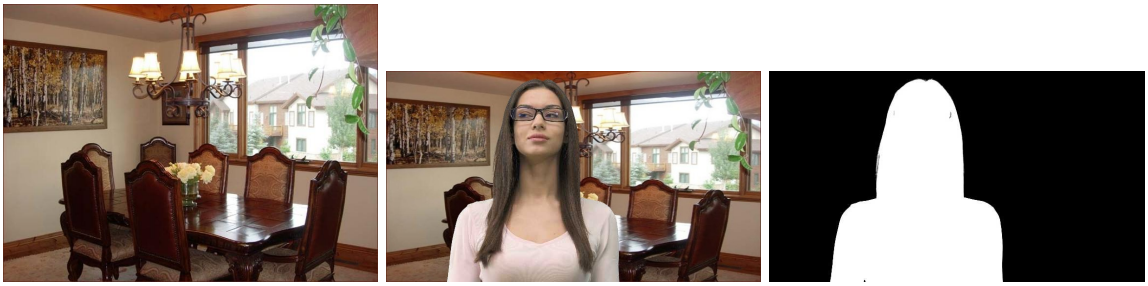
May 4th 2021

1 Problem statement

With the rise of working from home and attending meetings virtually, people are concerned about privacy concerns. When turning their camera on during an online event, the participants expose their background which might intrude on their personal life. To remediate this problem, most communication platforms adopted a background replacement technology where the users can display a virtual background instead of showing the room in which they are. While the idea is great and certainly solves the privacy issues, most companies did a mediocre job at extracting the background completely. This project focuses on perfecting background matting in real-time. Using two neural networks, the goal is to perform a high-resolution background replacement technique that operates at 30fps in 4K resolution, and 60fps for HD on a modern GPU.

2 Data Source

The input to our model consists of two images - a background image and a second image composed of a foreground, usually a human, posing in front of the same background (source). The goal of the model is to extract the human from the image. It predicts the alpha matte, a black and white image highlighting the foreground's shape. To train our model, we need three images per iteration: the background, the source, and the true alpha matte. There are public datasets consisting of high-quality green screen images that can be used to extract the alpha matte. There are also public background datasets but no combination of both. Therefore, using HD background images, Places365, and VideoMatte240K, we came up with our own dataset for this project. The image below shows the two input of the models (background and source) and the output it is trying to predict (alpha matte).



(a) (b) (c)

Figure 1: (a) the background (b) the source (c) the true alpha matte

The first step to creating our own dataset was to curate the background images. Since the main application of this project is to extract a person from their background during video calls, it is important to make sure that the background used as inputs to the model did not include any human. This denies the possibility of the model associating humans with backgrounds. Initially, we had over 30,000 background images so curating them manually would be time-consuming. We created a script that uses YOLO v3 weights [1] to delete the images with humans in them. You only look once (YOLO) is a state-of-the-art, object detection algorithm. It uses a single neural network and divides the full image into regions to perform object detection. This system classifies objects extremely fast which accelerated the process of creating our own dataset. The script deleted 7,000 images.

Figure 2: This shows the use of YoloV3 to create the dataset.

Using 23,312 backgrounds and 240,708 green screen images, we randomly paired an image from each set to form the source images. To avoid information loss, the background images were resized to the green screen images. To combine them, we used this simple formula on each pixel: $Finalimage = alpha * fgr + alphainv * bgr$ Where bgr, fgr, alpha, and alphainv are background, foreground, alpha matte, and inverse alpha matte respectively. Since alpha matte is in black and white (1 or 0 when normalized), multiplying one if its pixel to an image pixel either keeps the latter unchanged or deletes it. The foreground image is multiplied by the alpha matte which keeps the object of interest in the picture and deletes the rest of the foreground image. The background is multiplied by the inverse of the alpha matte which deletes every background pixel in the outline of the foreground object and keeps the rest. Below is a visual of the image blending process:

Figure 3: The process to create source images from background, foreground and alpha.

After blending, the source, background, and alpha matte have to be arranged in an ordered way to make it easier to feed the model.

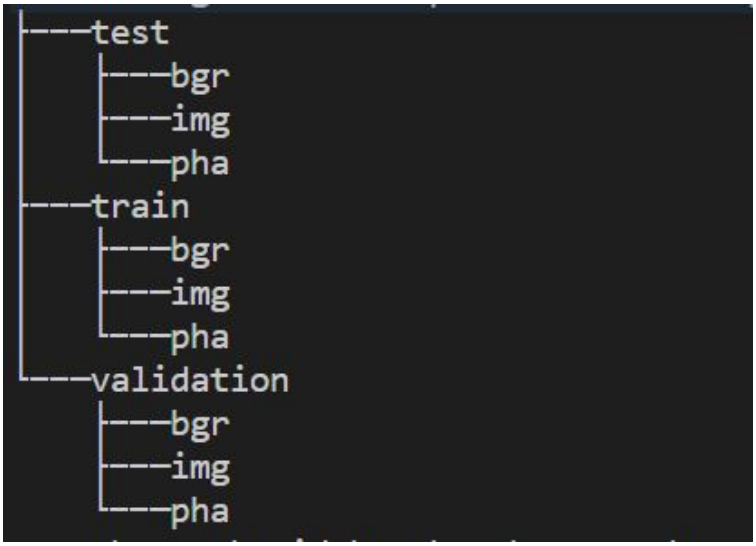


Figure 4: The directory tree of the dataset

The image above shows the directory tree of the dataset. Each image of a full set (background, source, alpha) is named the same and stored in their respective subfolder. This makes it easier to track down misplaced images. The data is split into 75/15/10 for training/validation/testing. It has 240,708 sets, so a total of 722,124 images. It can be accessed on the drive. Now that we have our curated dataset, we can go ahead and train our model

3 Model Architecture

We are implementing the architecture used by the paper[3] we are basing our project on. It consists of two models, a base and a refiner. The idea behind using two models is to have one produce a coarse alpha matte and the second model refine the challenging areas that the first model omitted. If the architecture used only one model, it would have to refine every pixel on the image to produce the same result quality. Since we are using the first model to produce a coarse alpha, the second model only has to refine around challenging areas (i.e around the hair) instead of the whole image. This reduces redundant computation and hence accelerates the background matting process while producing high-quality results. Let us have a deeper look at both models.

3.1 Base Network

The base network has two inputs - the background and the source - and four outputs, the coarse alpha, the foreground residual, the error map, and a hidden feature used in the refiner model. It is composed of three modules: a backbone, an ASPP, and a decoder. The backbone adopts ResNet-101 which imagenet weights to perform transfer learning. The Atrous Spatial Pyramid Pooling (ASPP) follows the DeepLabV3 approach and consists of multiple dilated convolution at different rates. The decoder applies bilinear upsampling at each layer and followed by 3x3 convolution, Batch Normalization, and ReLU activation. It then outputs the four features which are used in the refinement network.

3.2 Refinement Network

This model performs a first refinement by resampling the alpha matte, hidden features, and the input to the base network. Then, it uses the error map produced by the base network to select 4x4 patches with the highest predicted error to perform further refinement. 8x8 patches around these patches undergo two 3x3 convolutions with valid padding, Batch Normalization, and ReLu to reach the dimension of 4x4. They are then upsampled again to 8 x 8 and concatenated with the inputs of the base model at their respective location. Finally, the patches undergo two more 3x3 convolutions with valid padding, Batch Normalization, and ReLu. The refined areas are then swapped with their original patches. This network outputs a refined alpha and a foreground residual. Below is an illustration of the complete illustration.

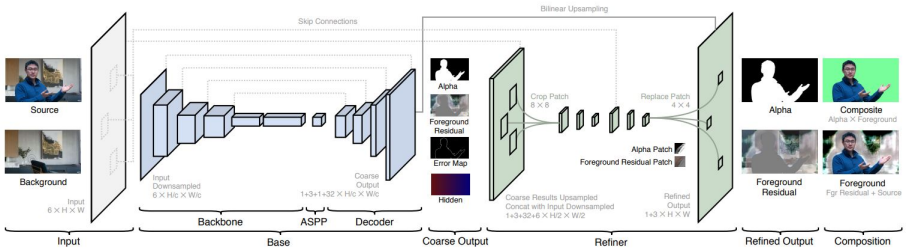


Figure 5: Illustration of the architecture offered by the Real-Time High-Resolution Background Matting paper [3]

4 Implementation Details

Implementing these dense network architectures proved a considerable challenge. The models described in the original researchers' paper are not representable with linear sequential models. Processing tensors from various stages in the model, residual skip connections, and parallel convolutional blocks are all present in these two models. Additionally, nearly every stage of the model outputs more than one tensor, often containing data of different shapes or data types. These features make it difficult to manage keeping track of the data in a useful way, as any improper conversion or missed concatenation can result in drastically worse model performance. Sometimes, as we found, those mistakes can also lead to dramatically inflated training time and computational expense. The remainder of this section will detail the implementation of each of those intricate parts.

4.1 Data Loading

Loading such a big dataset in the model has to be done in an organized way to avoid the possibility of running out of memory during training. Using the structure of the dataset directory and the ImageDataGenerator function, we could efficiently load the data by specifying its directory. Here is a snippet how we loaded the train images for training.

```
train_img_data = ImageDataGenerator(validation_split=TRAIN_SUBSET)
train_img_gen = train_img_data.flow_from_directory(
    os.path.normpath("./data/train/train_img"),
    target_size=IMAGE_DIMS,
    class_mode=None,
    subset='training',
    batch_size=BATCH_SIZE)
```

4.2 Functional Models

The complex nature these models inhibits the use of the Keras Sequential model. It is generally preferable to use this model, as it provides more safeguards against improperly connected tensors/layers. For these models, however, we made exclusive use of the Keras Functional API. The functional API differs from the standard sequential model in that any convolution, activation, or tensor operation can be used as stand-alone functions. More specifically, this means that models can be more parallel and more diverse with respect to its data. We see this first in the more involved use of a residual network (ResNet) in the base model.

4.3 Base Model Implementation

[2]

```
resnet = ResNet101(
    weights='imagenet',
    include_top=False,
    input_tensor=img
)
resnet.trainable=False
backbone_in = resnet.input
resblock1 = resnet.get_layer('conv1_relu').output
resblock2 = resnet.get_layer('conv2_block3_out').output
resblock3 = resnet.get_layer('conv3_block4_out').output
backbone_out = resnet.output
```

In the above snippet, we store various extracted features from the residual network by creating references to the outputs of the convolutional blocks. This gives the model information at multiple scales of analysis and from both simple and complex features. The ASPP structure also makes powerful use of the Functional API by analyzing the ResNet output in a pyramid form using increasing dilation rates. Then each layer of the pyramid is concatenated together to give a thorough multiple scale analysis to the decoder layer. The snippet below shows some of the ASPP structure. [4]

```
# conv block 1
conv1 = Conv2D(ASPP_FILTERS, 1, ...)(x)
conv1 = BatchNormalization(momentum=MOMENTUM, epsilon=EPSILON)(conv1)
conv1 = ReLU()(conv1)

# conv block 2
conv2 = Conv2D(ASPP_FILTERS, 3, ...)(x)
conv2 = BatchNormalization(momentum=MOMENTUM, epsilon=EPSILON)(conv2)
conv2 = ReLU()(conv2)

...

pyr = tf.concat([conv1,conv2,conv3,conv4,pool], axis=-1)
```

To conclude base model, the decoder network iteratively convolves and concatenates the layers of the pyramid with the residual blocks from the ResNet, allowing the network a more deep learning potential at each scale of analysis. It is worth noting here that we make use of the Keras Lambda layer to do gradient-safe tensor concatenation and resizing. Here is an example:

```
x = Lambda(lambda a: tf.image.resize(a, tf.shape(x2)[1:3]))(x)
```

The resulting output of the decoder section of the base model comprises 4 features contained in a 37-channel image. These features are the coarse alpha, foreground residual, error map, and hidden features. The coarse alpha makes up most of the model's final output. Because most of the matte is trivial to compute (large open areas in the corners of images), the coarse alpha does the easy part of the alpha prediction. The other 3 features, the foreground residual, error map, and hidden features, provide the necessary information for

the refinement network to process the more difficult parts of the images, namely, small things (hair, fingers) and translucent things (hair, some clothing). Those four features form the input tensor for the refinement model.

4.4 Refinement Model Implementation

To reiterate, the purpose of the refinement model is to clarify those pieces of the predicted alpha matte that are considerably more prone to error or more difficult to learn. The input tensor, in addition to the four features mentioned in the previous section, also contains the original source and background image. This allows the model to cleanly replace those portions of the image it deems part of the matte or outside of the matte. The researchers took multiple approaches to this part of the network. Their most involved image processing approach includes detecting those problematic sections of the error map (from the base model), performing high quality refinement on only those patches, and replacing those alpha and foreground patches with the higher resolution refinement result. This approach, which they refer to as the "full" refinement method, proved too difficult for us in the time allotted for this project. Instead, we opted to implement one of the less complex approaches, which they refer to as "sampling". First, we calculate some dimensions corresponding to the various scales at which the data gets sampled while being processed by the base model. Then, compare the produced coarse alpha matte from the base model with the feature maps and foreground maps through an iterative series of convolution and concatenation. This allows the model to compare the coarse alpha with higher and lower resolution images still, but without the more complex patching operations. By choosing this method, we have also sacrificed some degree of accuracy, seeing as the sampling method is a much more blunt tool than the full patching method.

4.5 Loss Functions

Two custom loss functions were written for training the base and refinement models. They perform image processing to ensure the same tensor shapes are present for the true and predicted alphas, and then pixel-wise mean-squared error loss is calculated. See the snippet below for the full implementation:

```
def base_alpha_mse_loss(pha_true, preds):
    pha_pred = tf.clip_by_value(preds[:, :, :, 0:1], 0, 1)
    pha_true = tf.cast(tf.reshape(pha_true, tf.shape(pha_pred)), tf.float32)
    return tf.math.squared_difference(pha_true, pha_pred)

def full_alpha_mse_loss(pha_true, pha_pred):
    pha_true = tf.cast(tf.reshape(pha_true, tf.shape(pha_pred)), tf.float32)
    return tf.math.squared_difference(pha_true, pha_pred)
```

5 Results

We trained the base model and refinement model separately. The base model trained on nearly 100000 images for 10 epochs at a batch size of 12. In total, the training took approximately 6 hours, with the resnet weights frozen. It is worth noting that because each sample requires 3 images, this means around 33000 samples in total. There was stable decrease in loss over the training period that did not indicate overfitting.

Across a variety of picture types, our model would correctly predict the coarse alpha values at an accuracy of 40-50%. When feeding a new image into the model, it can produce with reasonable accuracy the obvious areas where there is no human.

Additionally, one of the main motivations of this model is to achieve "real-time" background matting. In that vein, our model is able to produce predictions for a batch of 80 images in about 1.5 seconds. This means that, assuming a machine capable of displaying those images rapidly enough, our model can produce coarse mattes at nearly 60fps. This does not match the performance of the models in the paper, which are able to perform at 60fps for HD video, while ours is only operating on $768x432$ images.

6 Shortcomings and Future Improvements

The predicted alphas from our model are difficult to understand. In some images, the coarse alpha matte correctly matches the blank space around the human in the image, especially when the background image is particularly uniform. Unfortunately, when the background images are busier, the model struggles and produces very poor mattes. See the examples below:

Figure 6: An image where the model has confused the uniform color background image for the subject, superimposing it on the human.

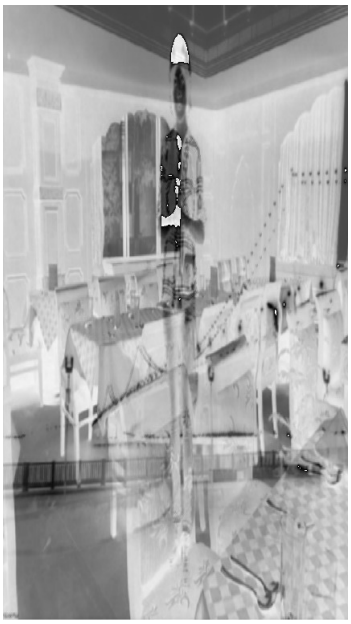


Figure 7: An image where the model has produced a clear matte of the human, but does not distinguish the background significantly.

In future work on this project, we will need to examine more closely the architecture of the model to find any areas where the hidden features are not distinguishing enough from the alpha. Currently, it is clear that the model has not been able to tell them apart accurately. Additionally, given the massive scale of the model itself, at over 60 million parameters, our training was simply inadequate to produce a robust model. A future improvement will involve far scaled training.

References

[1] A. Farhadi J. Redmon. *YOLOv3: A Incremental Improvement*. 2018. DOI: http://arxiv.org/abs/1804.02767.

[2] *Keras documentation: Developer guides*. URL: https://keras.io/guides/.

[3] Sengupta Lin Ryabtsev. *Real-Time High-Resolution Background Matting*. 2020. DOI: https://arxiv.org/pdf/2012.07810v1.pdf.

[4] *Tensorflow documentation*. URL: https://www.tensorflow.org/api_docs.