A. Overview

We provide additional details in this supplementary. In Sec. B, we describe the details of our network architecture and implementation. In Sec. C, we clarify our use of keywords for crawling background images. In Sec. D, we explain how we train our model and show details of our data augmentations. In Sec. E, we show additional metrics about our method’s performance. In Sec. F, we show all the qualitative results used in our user study along with the average score per sample.

B. Network

B.1. Architecture

**Backbone** Both ResNet and MobileNetV2 are adopted from the original implementation with minor modifications. We change the first convolution layer to accept 6 channels for both the input and the background images. We follow DeepLabV3’s approach and change the last downsampling block with dilated convolutions to maintain an output stride of 16. We do not use the multi-grid dilation technique proposed in DeepLabV3 for simplicity.

**ASPP** We follow the original implementation of ASPP module proposed in DeepLabV3. Our experiment suggests that setting dilation rates to (3, 6, 9) produces the better results.

*Equal contribution.

---

Figure 1: Network architecture. The diagram is repeated in supplementary for clarity. $G_{\text{base}}$ (blue) operates on the downsampled input to produce coarse-grained results and an error prediction map. $G_{\text{refine}}$ (green) selects error-prone patches and refines them to the full resolution.

---

**Decoder**

CBR128 - CBR64 - CBR48 - C37

"CBR$k$" denotes $k$ 3×3 convolution filters with same padding without bias followed by Batch Normalization and ReLU. "C$k$" denotes $k$ 3×3 convolution filters with same padding and bias. Before every convolution, decoder uses bilinear upsampling with a scale factor of 2 and concatenates with the corresponding skip connection from the backbone. The 37-channel output consists of 1 channel of alpha $\alpha$, 3 channels of foreground residual $F_{\text{R}}$, 1 channel of error map $E$, and 32 channels of hidden features $H$. We clamp $\alpha$ and $E$ to 0 and 1. We apply ReLU on $H$.

**Refiner**

First stage: C*BR24 - C*BR16
Second stage: C*BR12 - C*4

"C*BR$k$" and "C*$k$" follow the same definition above except that the convolution does not use padding.

Refiner first resamples coarse outputs $\alpha$, $F_{\text{R}}$, $H$, and input images $I$, $B$ to $\frac{1}{2}$ resolution and concatenates them as $[n \times 42 \times \frac{h}{2} \times \frac{w}{2}]$ features. Based on the error prediction $E$, we crop out top $k$ most error-prone patches $[nk \times 42 \times 8 \times 8]$. After applying the first stage, the patch dimension becomes $[nk \times 16 \times 4 \times 4]$. We upsample the patches with nearest
upsampling and concatenate them with patches at the corresponding location from $I$ and $B$ to form $[nk \times 22 \times 8 \times 8]$ features. After the second stage, the patch dimension becomes $[nk \times 4 \times 4 \times 4]$. The 4 channels are alpha and foreground residual. Finally, we bilinearly upsample the coarse $\alpha_c$ and $F^R_c$ to full resolution and replace the refined patches to their corresponding location to form the final output $\alpha$ and $F^R$.

**B.2. Implementation**

We implement our network in PyTorch [1]. The patch extraction and replacement can be achieved via the native vectorized operations for maximum performance. We find that PyTorch’s nearest upsampling operation is much faster on small-resolution patches than bilinear upsampling, so we use it when upsampling the patches.

**C. Dataset**

**VideoMatte240K** The dataset contains 484 video clips, which consists a total of 240,709 frames. The average frames per clip is 497.3 and the median is 458.5. The longest clip has 1500 frames while the shortest clip has 124 frames. Figure 2 shows more examples from VideoMatte240K dataset.

![Figure 2: More examples from VideoMatte240K dataset.](image)

**Background** The keywords we use for crawling background images are:

- airport interior
- bathroom
- church interior
- forest
- house outdoor
- lab interior
- mall interior
- rainy woods
- theater interior
- train station
- warehouse interior
- attic
- beach
- classroom interior
- garage interior
- interior
- night club interior
- rooftop
- stadium interior
- workplace interior
- bar interior
- city
- empty city
- gym interior
- kitchen
- lecture hall
- office
- theater interior
- warehouse interior

**D. Training**

Table 1 records the training order, epochs, and hours of our final model on different datasets. We use 1×RTX 2080 Ti when training only the base network and 2×RTX 2080 Ti when training the network jointly.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Network</th>
<th>Epochs</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>VideoMatte240K</td>
<td>$G_{base}$</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>VideoMatte240K</td>
<td>$G_{base} + G_{refine}$</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>PhotoMatte13K</td>
<td>$G_{base} + G_{refine}$</td>
<td>25</td>
<td>8</td>
</tr>
<tr>
<td>Distinctions</td>
<td>$G_{base} + G_{refine}$</td>
<td>30</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 1: Training epochs and hours on different datasets. Time measured on model with ResNet-50 backbone.

Additionally, we use mixed precision training for faster computation and less memory consumption. When using multiple GPUs, we apply data parallelism to split the mini-batch across multiple GPUs and switch to use PyTorch’s Synchronized Batch Normalization to track batch statistics across GPUs.

**D.1. Training augmentation**

For every alpha and foreground training sample, we rotate to composite with backgrounds in a “zip” fashion to form a single epoch. For example, if there are 60 training samples and 100 background images, a single epoch is 100 images, where the 60 samples first pair with the first 60 background images, then the first 40 samples pair with the rest of the 40 background images again. The rotation stops when one set of images runs out. Because the datasets we use are very different in sizes, this strategy is used to generalize the concept of an epoch.

We apply random rotation (±5deg), scale (0.3∼1), translation (±10%), shearing (±5deg), brightness (0.85∼1.15), contrast (0.85∼1.15), saturation (0.85∼1.15), hue (±0.05), gaussian noise ($\sigma^2 \leq 0.03$), box blurring, and sharpening independently to foreground and background on every sample. We then composite the input image using $I = \alpha F + (1 - \alpha) B$.

We additionally apply random rotation (±1deg), translation (±1%), brightness (0.82∼1.18), contrast (0.82∼1.18), saturation (0.82∼1.18), and hue (±0.1) only on the background 30% of the time. This small misalignment between input $I$ and background $B$ increases model’s robustness on real-life captures.

We also find creating artificial shadows increases model’s robustness because subjects in real-life often cast shadows on the environment. Shadows are created on $I$ by darkening some areas of the image behind the subject following the subject’s contour 30% of the time. Examples of composited images are shown in Figure 3. The bottom row shows examples of shadow augmentation.
D.2. Testing augmentation

For AIM and Distinctions, which have 11 human test samples each, we pair every sample with 5 random backgrounds from the background test set. For PhotoMatte85, which has 85 test samples, we pair every sample with only 1 background. We use the method and metrics described in [2] to evaluate the resulting sets of 55, 55, and 85 images.

We apply a random subpixel translation (±0.3 pixels), random gamma (0.85~1.15), and gaussian noise (μ = ±0.02, 0.08 ≤ σ² ≤ 0.15) to background B only, to simulate misalignment.

The trimaps used as input for trimap-based methods and for defining the error metric regions are obtained by thresholding the growth-truth alpha between 0.06 and 0.96, then applying 10 iterations of dilation followed by 10 iterations of erosion using a 3×3 circular kernel.

E. Performance

Table 2 shows the performance of our method on two Nvidia RTX 2000 series GPUs: the flagship RTX 2080 TI and the entry-level RTX 2060 Super. The entry-level GPU yields lower FPS but is still within an acceptable range for many real-time applications. Additionally, Table 3 shows that switching to a larger batch size and a lower precision can increase the FPS significantly.

F. Additional Results

In Figures 4, 5, 6, we show all 34 examples in the user study, along with their average rating and results by different methods. Figure 7 shows the web UI for our user-study.

References


### Table 2: Performance on different GPUs. Measured with batch size 1 and FP32 precision.

<table>
<thead>
<tr>
<th>GPU</th>
<th>Backbone</th>
<th>Reso</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTX 2080 TI</td>
<td>ResNet-50</td>
<td>HD</td>
<td>60.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4K</td>
<td>33.2</td>
</tr>
<tr>
<td></td>
<td>MobileNetV2</td>
<td>HD</td>
<td>100.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4K</td>
<td>45.4</td>
</tr>
<tr>
<td>RTX 2060 Super</td>
<td>ResNet-50</td>
<td>HD</td>
<td>42.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4K</td>
<td>23.3</td>
</tr>
<tr>
<td></td>
<td>MobileNetV2</td>
<td>HD</td>
<td>75.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4K</td>
<td>31.3</td>
</tr>
</tbody>
</table>

### Table 3: Performance using different batch sizes and precisions. Measured on RTX 2080 Ti.

<table>
<thead>
<tr>
<th>Backbone</th>
<th>Reso</th>
<th>Batch</th>
<th>Precision</th>
<th>FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MobileNetV2</td>
<td>HD</td>
<td>1</td>
<td>FP32</td>
<td>100.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>FP32</td>
<td>138.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>FP16</td>
<td>200.0</td>
</tr>
<tr>
<td></td>
<td>4K</td>
<td>8</td>
<td>FP16</td>
<td>64.2</td>
</tr>
</tbody>
</table>
Figure 4: Additional qualitative comparison (1/3). Average user ratings between Ours and BGM are included. A score of -10 denotes BGM is "much better", -5 denotes BGM is "slightly better", 0 denotes "similar", +5 denotes Ours is "slightly better", +10 denotes Ours is "much better". Our method receives an average 3.1 score.
Figure 5: Additional qualitative comparisons (2/3)
Figure 6: Additional qualitative comparisons (3/3)
Figure 7: The web UI for our user study. Users are shown the original image and two result images from Ours and BGM methods. Users are given the instruction to rate whether one algorithm is "much better", "slightly better", or both as "similar".