Supplementary Material *for*
Motion-Aware Dynamic Architecture for Efficient Frame Interpolation

In this supplementary document, we first show in Sec. 1 how the resource-aware regularization term in Eq. (11) can be decomposed into Eq. (14) in the main manuscript for our training curriculum. Then, we describe additional implementation details in Sec. 2 as well as the training/validation curves to analyze the training dynamics of the proposed framework in Sec. 3. We also show more detailed FLOPs/latency for each component of our model in Sec. 4, and demonstrate additional qualitative results in Sec. 5 and Sec. 6

1. Resource-aware loss decomposition

Recall the formula for calculating our resource-aware regularization term, Eq. (11) in the main manuscript, restated below:

$$\mathcal{R}_p = \sum_{i=1}^{n_s}\sum_{j=1}^{n_d} m^s_{p,i} \cdot m^d_{p,j} \cdot \mathcal{C}\left(\mathcal{H}(\cdot,\cdot\,;\, s_i, d_j)\right) \quad (1)$$

$$= \sum_{i=1}^{n_s}\sum_{j=1}^{n_d} m^s_{p,i} \cdot m^d_{p,j} \cdot \mathcal{C}\left(\mathcal{H}_{i,j}\right), \quad (2)$$

where we use the shortened notation $\mathcal{H}_{i,j} = \mathcal{H}(\cdot,\cdot\,;\, s_i, d_j)$ for simplicity. According to the second step of our training curriculum, we fix the scale to the original resolution, which we denote as $\mathcal{H}_{\hat{\mathbf{s}},j}$. Then, $\mathcal{C}\left(\mathcal{H}_{\hat{\mathbf{s}},j}\right)$ does not depend on the index i anymore, which let us to express $\mathcal{R}_p$ as

$$\mathcal{R}_p = \sum_{j=1}^{n_d} m^d_{p,j} \cdot \mathcal{C}\left(\mathcal{H}_{\hat{\mathbf{s}},j}\right) \cdot \sum_{i=1}^{n_s} m^s_{p,i} \quad (3)$$

$$= \sum_{j=1}^{n_d} m^d_{p,j} \cdot \mathcal{C}\left(\mathcal{H}_{\hat{\mathbf{s}},j}\right), \quad (4)$$

since $m^s_{p,i}$ is a one-hot vector and $\sum_{i=1}^{n_s} m^s_{p,i} = 1$. Restating our full loss function (Eq. (13) in the main manuscript) to region-wise loss function with the region index p, our loss function for the second training step becomes

$$\mathcal{L}_{total,p} = \mathcal{L}_{r,p} + \lambda_d \sum_{j=1}^{n_d} m^d_{p,j} \cdot \mathcal{C}\left(\mathcal{H}_{\hat{\mathbf{s}},j}\right), \quad (5)$$

where we rename the scaling hyperparameter λ to λ_d to emphasize that we are training for the DepthNet of our SD-finder.

Similarly, for the third training step, we denote our fixed-depth, varying-scale model as $\mathcal{H}_{i,\hat{\mathbf{d}}_j}$. Starting again from Eq. (2), we can reiterate Eq. (3)-(4) with interchanged scale and depth indices, so that

$$\mathcal{R}_p = \sum_{i=1}^{n_s} m^s_{p,i} \cdot \mathcal{C}\left(\mathcal{H}_{i,\hat{\mathbf{d}}_j}\right) \cdot \sum_{j=1}^{n_d} m^d_{p,j} \quad (6)$$

$$= \sum_{i=1}^{n_s} m^s_{p,i} \cdot \mathcal{C}\left(\mathcal{H}_{i,\hat{\mathbf{d}}_j}\right). \quad (7)$$

Then, the full loss function for the third training step can be expressed as

$$\mathcal{L}_{total,p} = \mathcal{L}_{r,p} + \lambda_s \sum_{i=1}^{n_s} m^s_{p,i} \cdot \mathcal{C}\left(\mathcal{H}_{i,\hat{\mathbf{d}}_j}\right), \quad (8)$$

where λ is again renamed to λ_s to emphasize that we are training for the ScaleNet of our SD-finder.

Combining Eq. (5) and (8), we can decompose Eq. (1) as

$$\mathcal{R}_p = \lambda_s \sum_{i=1}^{n_s} m^s_{p,i} \cdot \mathcal{C}\left(\mathcal{H}_{i,\hat{\mathbf{d}}}\right) + \lambda_d \sum_{j=1}^{n_d} m^d_{p,j} \cdot \mathcal{C}\left(\mathcal{H}_{\hat{\mathbf{s}},j}\right), \quad (9)$$

where $\lambda_s = 0$ for the second training curriculum, and $\lambda_d = 0$ for the third, respectively. This summarizes the detailed processes to obtain Eq. (14) of our main manuscript.

2. Additional implementation details

To modify the original CAIN [10] to have multiple exits, we did not use any additional layers but introduced an additional skip connection for each exit. For instance, if our SD-finder assigns the network depth to be one, 1) we compute the first group of residual blocks (Block1 in Fig. 2), 2) the output of Block1 passes through a new skip connection assigned to the first exit, which skips Block2-5 of CAIN, 3) we add the values contained in the global skip connection of CAIN, and 4) the output feature is passed through

Figure 1. Training dynamics for step 2 and step 3 of our training curriculum. We show the FLOPs, ℓ_1 loss ($\mathcal{L}_r$), and the full loss (Eq. (13) in the main manuscript) for training, and additionally the average PSNR for the validation set. Our final model is the one with the lowest (full) validation loss, which shows the best performance in the accuracy-resource trade-off curves.

the final convolution layer of CAIN and the PixelShuffle layer, which becomes the final output interpolation of the first exit. This process is similar to the 'intermediate image reconstruction (Fig. 6)' in the original CAIN paper, except that we compute the loss for all exits. Using this modified multi-exit CAIN, we train step 1 of our training curriculum with learning rate 1×10^{-5} and batch size 16 for 10k iterations. For step 2 and 3, we also fix the learning rate to 1×10^{-5}, but the batch size is set to fully utilize the GPU memory, which makes the minibatch size to be 8 for step 2 and 3 for step 3, respectively.

3. Training dynamics

We show the detailed training dynamics for our training step 2 and 3 in Fig. 1, where we train the DepthNet part of our SD-finder in step 2 and the ScaleNet part in step 3. Since the FLOPs loss is highly discretized, training is sometimes unstable; as visualized in the upper part of Fig. 1, there are cases when the validation FLOPs suddenly decrease significantly and the ℓ_1 loss increases correspondingly, and the increased overall loss cannot recover again with the current hyperparameters. We found that it is more tricky to train the DepthNet, which is why we learn the DepthNet first and fix its parameters for step 3. Note that, the proposed method can show better accuracy (PSNR) or more reduced FLOPs than the results reported in the main manuscript, but our final result is chosen to be the model with the lowest (full) validation loss, which implies that it is the best point on the accuracy-resource trade-off curve.

4. Detailed FLOPs / latency analysis

Table 1 shows the detailed FLOPs calculation and GPU latency measurements for each component in our proposed framework. For FLOPs, we can see that CAIN-D and

Table 1. Detailed FLOPs and GPU latency calculation for Xiph-2K videos.

	GFLOPs				GPU Latency (ms)			
	CAIN [10]	CAIN-S	CAIN-D	CAIN-SD	CAIN [10]	CAIN-S	CAIN-D	CAIN-SD
Flow estimation	-	22.55	22.55	22.55	-	25.5	25.5	25.5
SD-finder	-	1.28	1.42	2.71	-	0.6	0.5	1.1
Interpolation	3,132.78	3620.19	1,674.92	1,424.54	224.6	201.0	169.7	138.8
Super-resolution	-	146.95	-	148.64	-	65.1	-	71.3
Total	3,132.78	3,790.97	1,698.89	1,598.44	224.6	292.2	195.7	236.7

CAIN-SD greatly reduces the number of operations required for the 'Interpolation' part. Since we use the downscaled input frames for optical flow estimation, the required FLOPs for 'Flow estimation' and 'SD-finder' is almost negligible. Also, the 'Super-resolution' part does not require that many operations, which tells us that interpolation model is the bottleneck in FLOPs consumption.

For the GPU latency, although optical flow estimation or the super-resolution parts do not require many FLOPs, they still have many layers that have to be sequentially passed through, which lead to considerable running time. We can see that CAIN-D and CAIN-SD significantly reduce the running time of the interpolation model compared to the baseline CAIN, which lead to Since SD-finder is composed of two very lightweight 3-layer CNNs, the actual latency requirement is negligible in the overall latency.

5. Additional Qualitative Results

Fig. 2 and 3 depicts the additional qualitative results that are analogous to Fig. 3 and 4 in the main paper. From Fig. 2, we can see that the PSNR values for the regions that have relatively plain textural details or the background regions that have small motion are similar regardless of the model depth, and our SD-finder chooses to exit early to save computation. On the other hand, complex regions with large motion are passed through deeper layers, while some regions find a compromise in between and choose to exit in the middle ($d =$Ⓓ).

In Fig. 3, we demonstrate the additional qualitative results with the scale predictions. Similar to the results shown in the main paper, $s = 1$ or $s = 2$ is usually chosen and $s = 4$ is not due to the notable performance degradation. Note that, for some regions, downscaling and upscaling helps to find the correct position of the moving objects and the PSNR for $s = 2$ is higher than the original scale, even with less computation cost.

6. Video Demo

Please refer to our project page[1] for more qualitative results and the videos on full-frame comparisons that aggregate all patches. There exists some boundary effects between the borders of the patches that contain extremely large motion, but the visual quality of our proposed CAIN-SD model is competitive with the full CAIN in general.

[1] https://myungsub.github.io/adaptive-int

Figure 2. Comparison between the local regions with different depth predictions. $d =$Ⓔ represents the full model inference (original CAIN), while $d =$Ⓐ shows the output at the first early exit. Blue box shows the "easy" regions where the model already performs well on its first exit and saves excessive computations. On the other hand, deeper layers are needed to maintain the performance for more complex regions with the green box, where our model decides to pass through the full interpolation model.

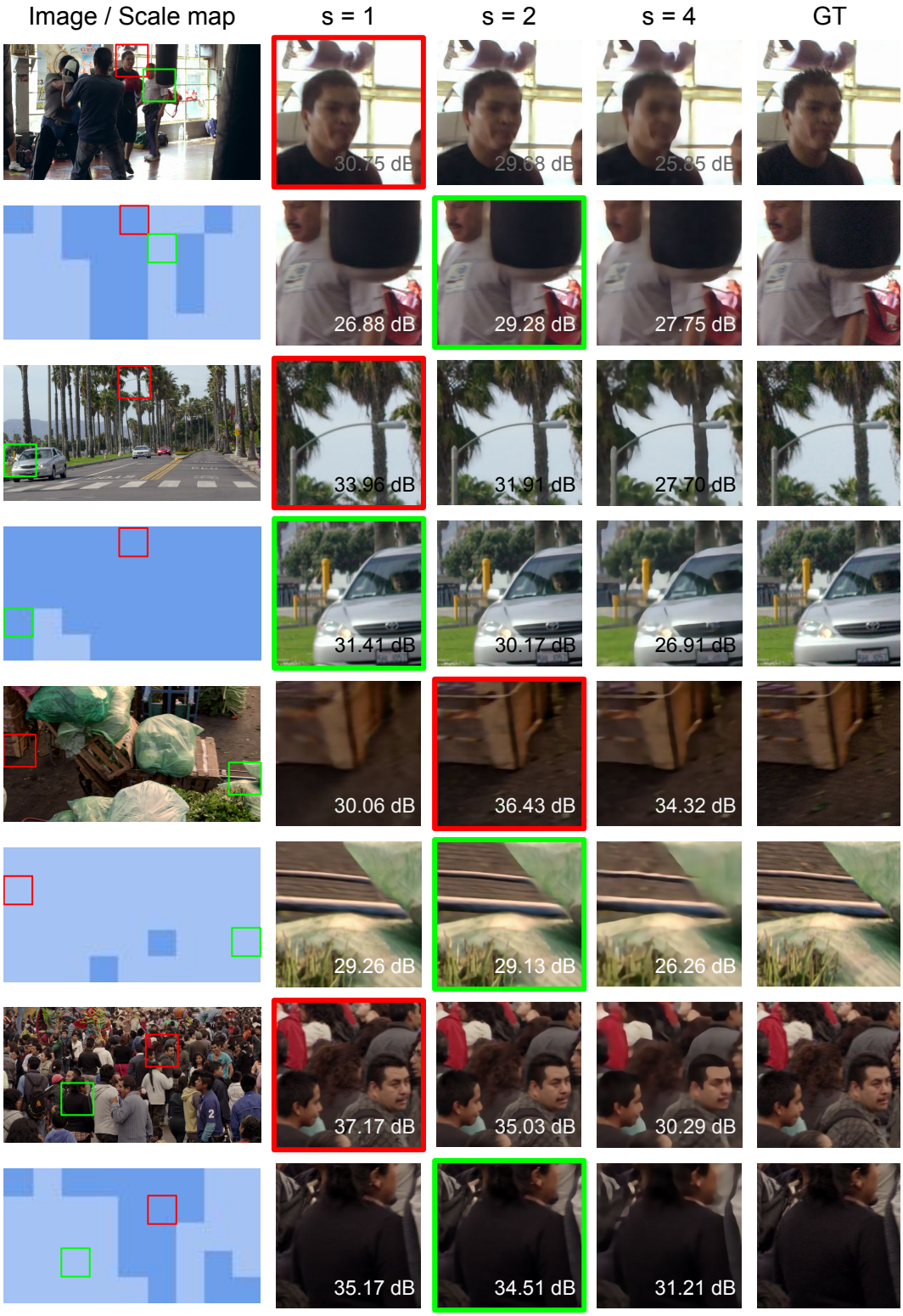

Figure 3. Comparison between the regions with different scale predictions. Regions with the red box significantly lose the textural details when the input frames are downscaled, so the original scale is used to maintain the performance. Green box shows the regions where our SD-finder chooses the scaling factor of 2, where we can save computations compared to $s = 1$ with even increased PSNR.