Supplementary Materials for Fast Bi-layer Neural Synthesis of Realistic Head Avatars

Egor Zakharov[1,2], Aleksei Ivakhnenko[1], Aliaksandra Shysheya[1,3], and Victor Lempitsky[1,2]

[1] Samsung AI Center – Moscow, Russia
[2] Skolkovo Institute of Science and Technology, Russia
[3] University of Cambridge, UK

1 Methods

We start by explaining training process of our method in much more details. Then, we describe the architecture that we use and how different choices affect the final performance. Finally, we provide a more extended explanation of the mobile inference pipeline that we have adopted.

1.1 Training details

We optimize all networks using Adam [5] with a learning rate equal to $2 \cdot 10^{-4}$ $\beta_1 = 0.5$ and $\beta_2 = 0.999$. Before testing, we calculate "standing" statistics for all batch normalization layers using 500 mini-batches. Below we provide additional details for the losses that we use.

Texture mapping regularization. Below we provide additional implementation details as well as better describe the reasons why this loss is used.

The training signal that the texture generator G_{tex} receives is first warped by the warping field $\omega^i(t)$ predicted by the inference generator. Because of this, random initializations of the networks typically lead to subpotimal textures, in which the face of the source person occupies a small fraction of the total area of the texture. As the training progresses, this leads to a lower effective resolution of the output image, since the optimization process is unable to escape this bad local optima.

In practice, we address the problem by treating the network's output as a delta to an identity mapping, and also by applying a magnitude penalty on that delta in the early iterations. As mentioned in the main paper, the weight of this penalty is multiplicatively reduced to zero during training, so it does not affect the final performance of the model. More formally, we decompose the output warping field into a sum of two terms: $\omega^i(t) = \mathcal{I} + \Delta\omega^i(t)$, where $\mathcal{I}$ denotes an identity mapping, and apply an L_1 penalty, averaged by a number of spatial positions in the mapping, to the second term:

$$\mathcal{L}^G_{\text{reg}} = \frac{1}{HW} ||\Delta\omega^i(t)||_1 \,. \tag{1}$$

To understand why this regularization helps, we need to briefly describe the implicit properties of the VoxCeleb2 dataset. Since it was obtained using a face detector, a weak from of face alignment is present in the training images, with face occupying more or less the same region.

On the other hand, our regularization allows the gradients to initially flow unperturbed into the texture generator. Therefore, gradients with respect to the texture, averaged over the minibatch, consistently force the texture to produce a high-frequency component of a mean face in the minibatch. This allows the face in the texture to fill the same area as it does in the training images, leading to better generalization.

Adversarial loss. Below we elaborate in more details on the type of adversarial loss that is used. We use the terms (2) and (3) to calculate realism scores for real and fake images respectively, with i_n and t_n denoting indices of mini-batch elements, N – a mini-batch size and $i \in \{i_1, \dots, i_n\}$:

$$\mathbf{s}^i(t) = D\big(\mathbf{x}^i(t), \mathbf{y}^i(t)\big) - \frac{1}{N}\sum_n^N D\big(\hat{\mathbf{x}}^{i_n}(t_n), \mathbf{y}^{i_n}(t_n)\big)\,, \tag{2}$$

$$\hat{\mathbf{s}}^i(t) = D\big(\hat{\mathbf{x}}^i(t), \mathbf{y}^i(t)\big) - \frac{1}{N}\sum_n^N D\big(\mathbf{x}^{i_n}(t_n), \mathbf{y}^{i_n}(t_n)\big)\,. \tag{3}$$

Moreover, we use PatchGAN [4] formulation of the adversarial learning. In it, the discriminator outputs a matrix of realism scores instead of a single prediction, and each element of this matrix is treated as a realism score for a corresponding patch in the input image. This formulation is also used in a large body of relevant works [3,7,8] and improves the stability of the adversarial training. If we denote the size of a scores matrix $\mathbf{s}^i(t)$ as $H_s \times W_s$, the resulting objectives can be written as follows:

$$\mathcal{L}^D_{\text{adv}} = \frac{1}{H_s W_s}\sum_{h,w} \max\big(0, 1 - \mathbf{s}^i_{h,w}(t)\big) + \max\big(0, 1 + \hat{\mathbf{s}}^i_{h,w}(t)\big)\,, \tag{4}$$

$$\mathcal{L}^G_{\text{adv}} = \frac{1}{H_s W_s}\sum_{h,w} \max\big(0, 1 + \mathbf{s}^i_{h,w}(t)\big) + \max\big(0, 1 - \hat{\mathbf{s}}^i_{h,w}(t)\big)\,. \tag{5}$$

The loss (4) serves as the discriminator objective. For the generator, we also calculate the feature matching loss [8], which has now become a standard component of supervised image-to-image translation models. In this objective, we minimize the distance between the intermediate feature maps of discriminator, calculated using corresponding target and generated images. If we denote as $\mathbf{f}^i_{k,D}(t)$ the features at different spatial resolutions $H_k \times W_k$, then the feature mathing objective is computed as follows:

$$\mathcal{L}^G_{\text{FM}} = \frac{1}{K}\sum_k \frac{1}{H_k W_k} ||\hat{\mathbf{f}}^i_{k,D}(t) - \mathbf{f}^i_{k,D}(t)||_1\,. \tag{6}$$

1.2 Architecture description

All our networks consist of pre-activation residual blocks. The layout is visualized in the Figures 1-5. In all networks, except for the inference generator at the updater, we set the minimum number of channels to 64, and increase (decrease) it by a factor of two each time we perform upsampling (downsampling). We pick the first convolution in each block to increase (decrease) the number of channels. The maximum number of channels is set to 512. In the inference generator we set the minimum number of channels to 32, and the maximum to 256. Also, all linear layers (except for the last one) have their dimensionality set to 256. Moreover, as described in Figure 2, in the inference generator we employ more efficient blocks, with upsampling performed after the first convolution, and not before it. This allows us to halve the number of MACs per inference.

In the embedder network (Figure 3) each block operating at the same resolution reduces the number of channels, similarly to what is done in the generators. In fact, the output number of channels in each block is excatly equal to the input number of channels in the corresponding generator block. We borrowed this scheme from [7], and assume that is it done to botteleneck the embedding tensors, which will be used for the prediction of the adaptive parameters at high resolution. This forces the generators to use all their capacity to generate the image bottom-up, instead of using a shortcut between the source and the target at high resolution, which is present in the architecture.

We do not use batch normalization in the embedder network, because we want it to be trained more slowly, compared to other networks. Otherwise, the whole system overfits to the dataset and the textures become correlated with the source image in terms of head pose. We believe that this is related to the VoxCeleb2 dataset, since in it there is a strong correlation in terms of pose between the randomly sampled source and target frames. This implies that the dataset is lacking diversity with respect to the head movement, and we believe that our system would perform much better either with a better disentangling mechanism of head pose and identity, which we did not come up with, or with a more diverse dataset.

On contrary, we find it highly beneficial to use batch normalization in the discriminator (Figure 4). This is less memory efficient, compared to the classical scheme, since “real” and “fake” batches have to be concatenated and fed into the discriminator together. We concatenate these batches to ensure that the first and second order statistics inside the discriminator’s features are not whitened with respect to the label (“real” or “fake”), which significantly improves the quality of the outputs.

We also tried using instance normalization, but found this to be more sensitive to hyperparameters. For example, the config working on a high-quality dataset cannot be transferred to the low-quality dataset without the occurring instabilities during the adversarial training.

We predict adaptive parameters following the procedure inspired by a matrix decomposition. The basic idea is to predict a weight tensor for the convolution via a decomposition of the embedding tensor. In our work, we use the following

procedure (taken from [7]) to predict the weights for all 1×1 convolutions and adaptive batch normalization layers in the texture and the inference generators:

- Resize all embedding tensors $\hat{\mathbf{e}}_k^i(s)$, with the number of channels C_k, by nearest upsampling to 32×32 resolution for the texture generator, and 16×16 for the medium-sized inference generator.

- Flatten the resized tensor across its spatial resoluton, converting it to a matrix of the shape $C_k \times 1024$ for the texture generator, and $\frac{1}{2}C_k \times 512$ for the inference generator (the first dimensionality has to match the reduced number of channels in the convolutions of the medium-sized model).

- Three linear layers (with no nonlinearities in between) are then applied, performing the decomposition. A resulting matrix should match the shape of the weights, combined with the biases, for each specific adaptive layer. These linear layers are trained separately for each adaptive convolution and adaptive batch normalization.

Each embedding tensor $\hat{\mathbf{e}}_k^i(s)$ is therefore used to predict all adaptive parameters inside the layers of the k-th block in the texture and inference generators. We do not perform an ablation study with respect to this scheme, since it was used in an already published work on a similar topic.

Finally, we describe the architecture of the texture enhancer in Figure 5. This architecture is standard for image-to-image translation tasks. The spatial dimensionality and the number of channels in the bottleneck is equal to 128.

1.3 Mobile inference

As mentioned in main paper, we train our models using PyTorch and then port them to smartphones with Qualcomm Snapdragon 855 chips. For inference, we use a native Snapdragon Neural Processing Engine (SNPE) APK, which provides a significant speed-up compared to TF-Lite and PyTorch mobile. In order to convert the models trained in PyTorch into SNPE-compatible containers, we first use the PyTorch-ONNX parser, as it is simple to get an ONNX model right from PyTorch. However, it does not guarantee that the obtained model can be converted into a mobile-compatible container, since some operations may be unsupported by SNPE. Moreover, there is a collision between different versions of ONNX and SNPE operation sets, with some versions of the operations being incompatible with each other. We have solved this problem by using PyTorch 1.3 and SNPE 1.32, but solely for operations used our inference generator. This is part of the reason why we had to resort to simple layers, like BathNorm-s, convolutions and nonlinearities in our network..

All ported models have spectral normalization removed, and adaptive parameters fixed and merged into their base layers. In our experiments the target platform is Adreno 640 GPU, utilized in FP16 mode. We do not observe any noticeable quality degradation from running our model in FP16 (although training

in FP16 or mixed precision settings leads to instabilities and early explosion of the gradients). Since our model includes bilinear sampling from texture (using a predicted warping field), that is not supported by SNPE, we implement it ourselves, as a part of application, called after each inferred frame on a CPU. The GPU implementation should be possible as well, but is more time-consuming to implement. Our reported mobile timings (42 ms, averaged by 100 runs) do not include the bilinear sampling and copy operations from GPU to CPU. On CPU, bilinear sampling takes additional 2 milliseconds, but for a GPU implementation, the timing would be negligible.

2 Experiments

2.1 Training details for the state-of-the-art methods.

First Order Motion model was trained using a config provided with the official implementation of the model. In order to obtain a family of models, we modify minimum and maximum number of channels in the generator from default 64 and 512 to 32 and 256 for the medium, and 16 and 128 for the small models.

For Few-shot Vid-to-Vid, we have also used a default config from the official implementation, but with slight modifications. Since we train on a dataset with videos already being cropped, we removed the random crop and scale augmentations in order to avoid a domain gap between training and testing. In our case, that would lead to black borders appearing on the training images, and a suboptimal performance on a test set with no such artifacts. In order to obtain a family of models, we also reduce the minimum and maximum number of channels in the generator from the default 32 and 1024 to 32 and 256 for the medium model and 16 and 128 for the small model.

To calculate the number of multiply-accumulate operations, we used an off-the-shelf tool that evaluates this number for all internal PyTorch modules. That way of calculation, while being easy, is not perfect as, for example, it does not account for the number of operations in PyTorch functionals, which may be called inside the model. Other forms of complexity evaluation would require significant refactor of the code of the competitors, which lies out of the scope of our comparison. For our model, we have provided accurate complexity esimates.

2.2 Extended evaluations.

We provide extended quantitative data for our experiments in Table 1, and additional qualitative comparisons in Figures 6-8, which extend the comparisons provided in the main paper. We additionally perform a small comparison with a representative mesh-based avatar system [1] in Figure 9 and compare our method with MarioNETte system [3] in Figure 11. Also we extend our ablation study to highlight the contribution of the texture enhancement network in the Figure 10. Finally, we show cross-person reenactment results in Figure 12.

Method	LPIPS↓	SSIM↑	CSIM↑	NME↓	GMACs↓	Init. (ms)↓	Inf. (ms)↓
Small models							
F-s V2V	0.389	-	0.600	0.581	10.2	-	-
FOMM	**0.325**	-	**0.622**	0.503	3.78	-	-
Ours	0.392	-	0.540	**0.475**	**1.08**	-	-
Medium models							
F-s V2V	0.368	0.419	0.604	0.461	18.2	4	22
FOMM	**0.311**	**0.553**	0.638	0.478	13.9	**3**	13
Ours	0.358	0.508	**0.653**	**0.433**	**4.32**	53	**4**
Large models							
NTH	0.386	-	0.419	0.459	52.8	-	-
F-s V2V	0.364	-	0.623	0.441	22.2	-	-
FOMM	**0.298**	-	**0.661**	0.450	53.7	-	-
Ours	0.356	-	0.655	**0.428**	**17.3**	-	-

Table 1: We present numerical data for the comparison of the models. Some of it duplicates the data available in Figure 5 of the main paper. F-s V2V denotes Few-shot Vid-to-Vid [7], FOMM denotes First Order Motion Model [6], and NTH denotes Neural Talking Heads [9]. Here we also include SSIM evaluation, which we found to correlate with LPIPS, and therefore excluded it from the main paper. We also provide evaluation for initialization and inference time (in milliseconds) for the medium-sized models of each method, measured on NVIDIA P40 GPU. We did not include this measurement in the main paper since we cannot calculate it using target low-performance devices (due to difficulties with porting the competitor models to the SNPE [2] framework), while evaluation on much more powerful (in terms of FLOPs) desktop GPUs may be an inaccurate way to measure the performance on less capable devices. We, therefore, decided to stick with MACs as our performance metric, which is more common in the literature, but still provide our obtained numbers for desktop GPUs here. We report median values out of a thousand iterations with random inputs.

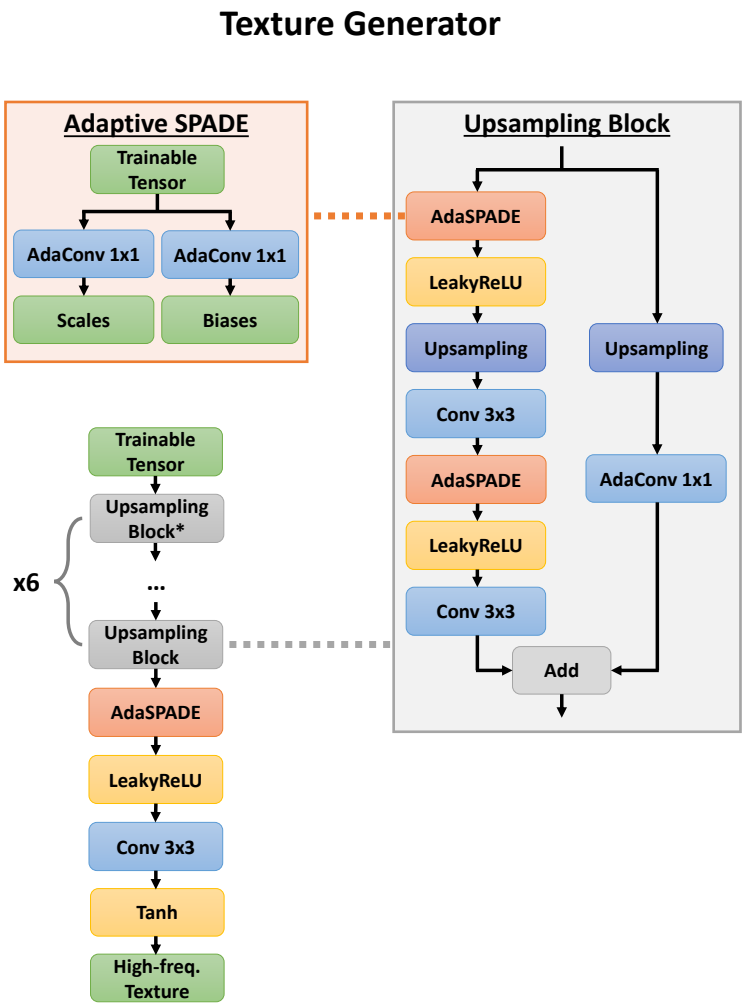

Fig. 1: Description of the texture generator's architecture. The first normalization layer in the first upsampling block (marked with a star) is replaced by a regular batch normalization. For the spatial resolution increase, nearest upsampling is performed. All trainable tensors in adaptive SPADE layers have the same size as an output of the previous layer. The first trainable tensor, which is a network's input, has a spatial resolution of 4×4.

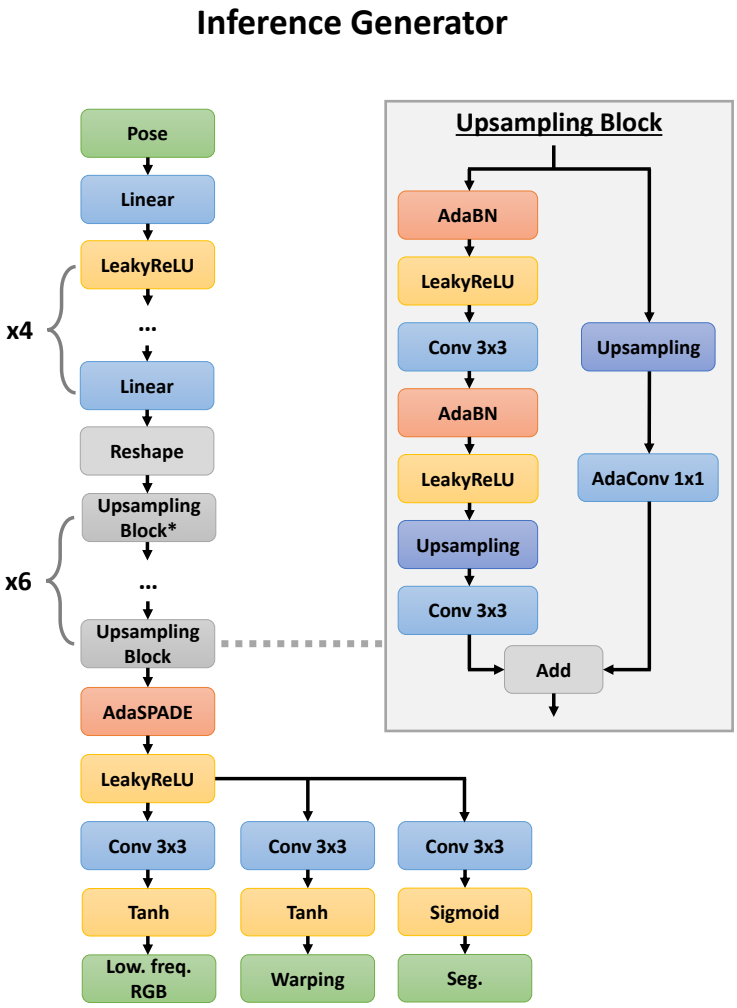

Fig. 2: The architecture of the inference generator. As in the texture generator, in the first upsampling block the first normalization layer is replaced by a regular batch normalization. Similarly, nearest upsamping is used. Input pose is reshaped into a vector and fed into a stack of linear layers. Then, the output of the last linear layer is reshaped to have a spatial resolution of 4×4.

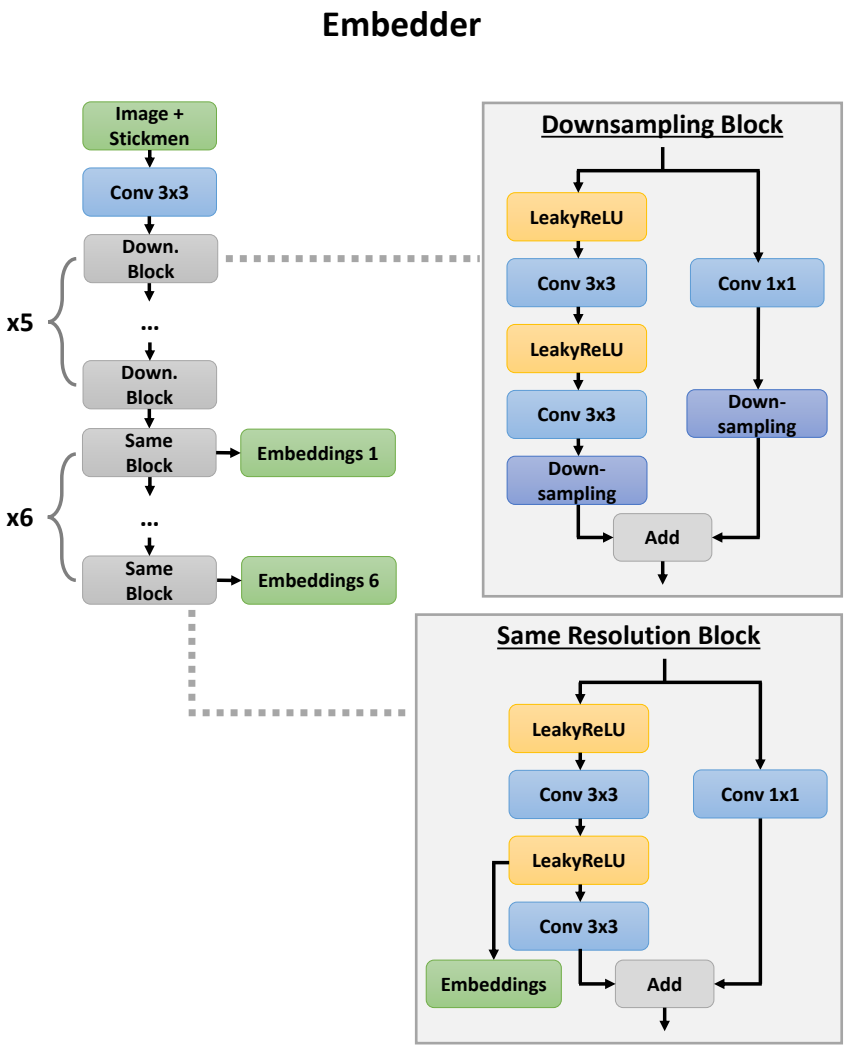

Fig. 3: Architecture for the embedder. Here we do not use normalization layers. First, we downsample input images and stickmen to 8×8 resolution. After that, we obtain embeddings for each of the blocks in the texture and the inference generators. Each embedding is a feature map, and has the same number of channels as the corresponding block in the texture generator. Therefore, we reduce the number of channels in the final blocks, from the maximum of 512 to the minimum of 64 at the end. In the blocks operating at the same resolution, we insert a convolution into a skip connection only when the input and the output number of channels is different.

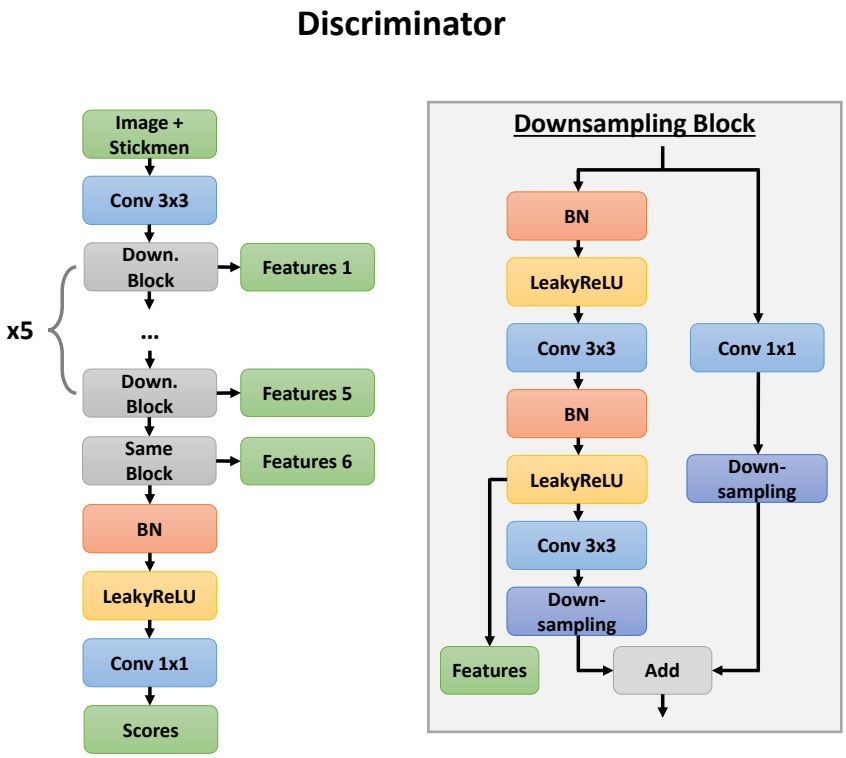

Fig. 4: Architecture of the discriminator. We use 5 downsampling blocks and one block operating at final 8×8 resolution. Additionally, in each block we output features after the second nonlinearity. These features are later used in the feature matching loss. For downsampling, we use average pooling. The architecture of the final block, operating at the same resolution, is similar to the one in the embedder: it is without a convolution in the skip connection, but with batch normalization layers.

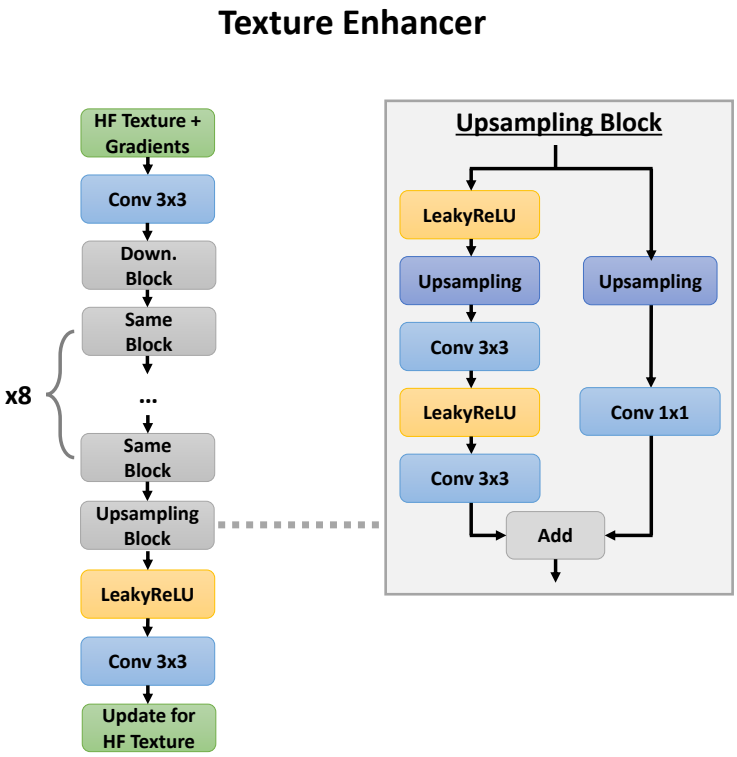

Fig. 5: We employ a simple encoder-decoder style architecture, similar to the one used in [4]. We replace downsampling and upsampling layers with residual blocks. We also do not employ batch normalization inside the enhancer.

Fig. 6: Extended comparison of the medium-sized models from all method families on the VoxCeleb2 dataset. For Few-shot Talking Heads we use the results obtained using the original full-sized model.

Fig. 7: Detailed qualitative results for our medium-sized model trained on the VoxCeleb2-HQ dataset.

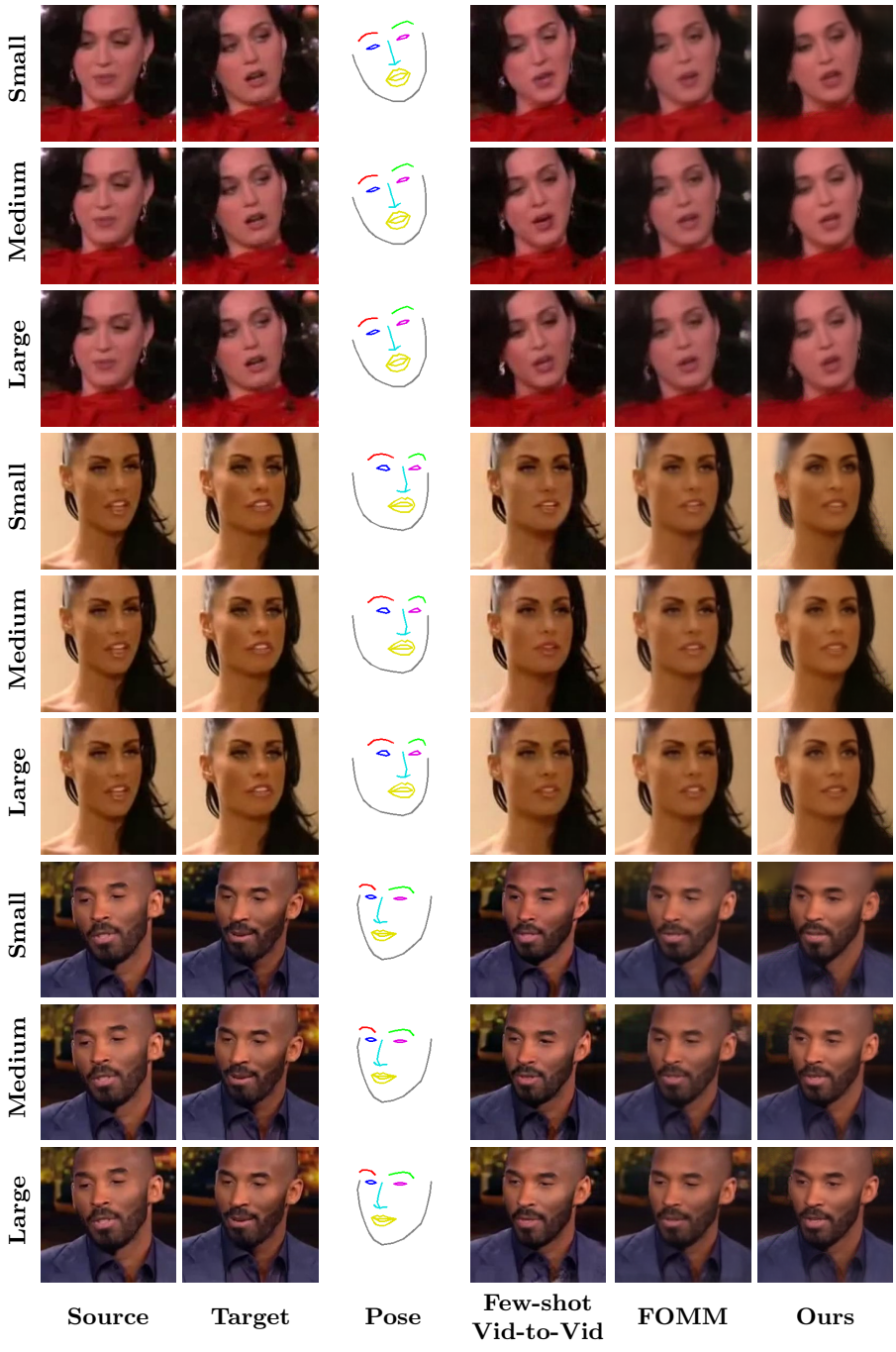

Fig. 8: Qualitative comparison between the small, medium and large models for all compared families of methods.

Fig. 9: Comparison of our method with a closed-source product [1], which is representative of the state-of-the-art in real-time one-shot avatar creation, based on explicit 3D modelling. The first row represents reenactment results, since the frontal image was used for initialization of both methods. We can see that our model does a much better job of modelling the face shape and the hair.

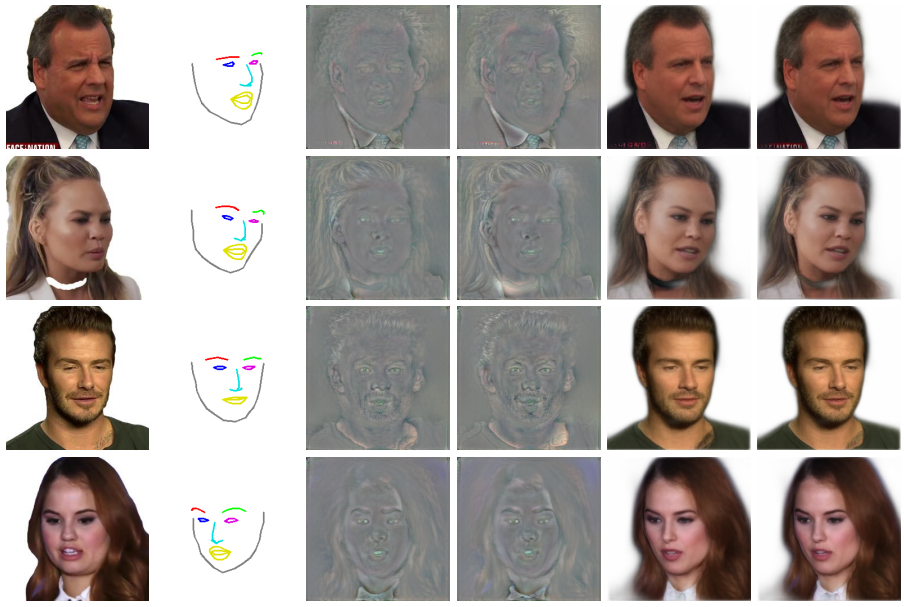

Fig. 10: Ablation study for the contribution of the texture updater on a VoxCeleb2-HQ dataset. The results are presented with and without the updater.

Fig. 11: A comparison with MarioNETte [3] system in a one-shot self-reenactment task. The results for [3] are taken from the respective paper, as no source code is available. The evaluation of the computational complexity of this system was also beyond our reach since it would require re-implementation from scratch. However, since it utilizes an encoder-decoder architecture with a large number of channels [3], it can be assumed to have a similar complexity to the largest variant of FOMM [6]. For our method, we use a medium-sized model. Lastly, the evaluation for [3] is done on the same videos as training (on the hold-out frames), while our method is applied without any fine-tuning.

Fig. 12: The results for cross-person reenactment. While our method does preserve the texture of the original image, the driving identity leakage remains noticeable.

References

1. Avatar SDK homepage. https://avatarsdk.com
2. SNPE homepage. https://developer.qualcomm.com/sites/default/files/docs/snpe
3. Ha, S., Kersner, M., Kim, B., Seo, S., Kim, D.: Marionette: Few-shot face reenactment preserving identity of unseen targets. CoRR **abs/1911.08139** (2019)
4. Isola, P., Zhu, J., Zhou, T., Efros, A.A.: Image-to-image translation with conditional adversarial networks. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017 (2017)
5. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. CoRR **abs/1412.6980** (2014)
6. Siarohin, A., Lathuilière, S., Tulyakov, S., Ricci, E., Sebe, N.: First order motion model for image animation. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019 (2019)
7. Wang, T., Liu, M., Tao, A., Liu, G., Catanzaro, B., Kautz, J.: Few-shot video-to-video synthesis. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019 (2019)
8. Wang, T., Liu, M., Zhu, J., Tao, A., Kautz, J., Catanzaro, B.: High-resolution image synthesis and semantic manipulation with conditional gans. In: 2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018 (2018)
9. Zakharov, E., Shysheya, A., Burkov, E., Lempitsky, V.S.: Few-shot adversarial learning of realistic neural talking head models. In: IEEE International Conference on Computer Vision, ICCV 2019 (2019)