A. Limitations and potential negative impact

A.1. Limitations

Our model has the following limitations:

- *Limitations of sparse training*. In general, sparse training makes it impossible for D to capture complex dependencies between frames. But surprisingly, it provides state-of-the-art results on modern datasets, which (using the statement from §3.3) implies that they are not that sophisticated in terms of motion.
- *Dataset-induced limitations*. Similar to other machine learning models, our method is bound by the dataset quality it is trained on. For example, for FaceForensics 256^2 dataset [54], our embedding and manipulations results are inferior to StyleGAN2 ones [1]. This is due to the limited number of identities (just 700) in FaceForensics and their larger diversity in terms of quality compared to FFHQ [30], which StyleGAN2 was trained on.
- *Periodicity artifacts*. G still produces periodic motions sometimes, despite of our acyclic positional embeddings. Future investigation on this phenomena is needed.
- *Poor handling of new content appearing*. We noticed that our generator tries to reuse the content information encoded in the global latent code as much as possible. It is noticeable on datasets where new content appears during a video, like Sky Timelapse or Rainbow Jelly. We believe it can be resolved using ideas similar to ALIS [62].
- *Sensitivity to hyperparameters*. We found our generator to be sensitive to the minimal initial period length $\max_i \sigma_i$ (See Appx B). We increased it for SkyTimelapse [79] from 16 to 256: otherwise it contained unnatural sharp transitions.

We plan to address those limitations in our future works.

A.2. Potential negative impact

The potential negative impact of our method is similar to those of traditional image-based GANs: creating "deepfakes" and using them for malicious purposes.[6]. Our model made it much easier to train a model which produces much more realistic video samples with a small amount of computational resources. But since the availability of high-quality datasets is very low for video synthesis, the resulted model will fall short compared to its image-based counterpart, which could use rich, extremely qualitative image datasets for training, like FFHQ [30].

[6] https://en.wikipedia.org/wiki/Deepfake

B. Implementation and training details

Note, that all the details can be found in the source code: https://github.com/universome/stylegan-v.

B.1. Optimization details and hyperparameters

Our model is built on top of the official StyleGAN2-ADA [29] repository[7]. In this work, we build a model to generate continuous videos and a reasonable question to ask was why not use INR-GAN [61] instead (like DI-GAN [5]) to have fully continuous signals? The reason why we chose StyleGAN2 instead of INR-GAN is that StyleGAN2 is amenable to the mixed-precision training, which makes it train $\approx$2 times faster. For INR-GAN, enabling mixed precision severely decreases the quality and we hypothesize the reason if it is that each pixel in INR-GAN's activations tensor carries more information (due to the spatial independence) since the model cannot spatially distribute information anymore. And explicitly restricting the range of possible values adds a strict upper bound on the amount of information one each pixel is able to carry. We also found that adding coordinates information does not improve video quality for our generator neither qualitatively, nor in terms of scores.

Similar to StyleGAN2, we utilize non-saturating loss and R_1 regularization with the loss coefficient of 0.2 in all the experiments, which is inherited from the original repo and we didn't try any hyperparameter search for it. We also use the `fmaps` parameter of 0.5 (the original StyleGAN2 used fmaps parameter of 1.0), which controls the channel dimensionalities in G and D, since it is the default setting for StyleGAN2-ADA for 256^2 resolution. This allowed us to further speedup training.

The dimensionalities of $\boldsymbol{w}, \boldsymbol{z}, \boldsymbol{u}_t, \boldsymbol{v}_t$ are all set to 512.

As being stated in the main text, we use a padding-less `conv1d`-based motion mapping network F_m with a large kernel size to generate raw motion codes $\boldsymbol{u}_t$. In all the experiments, we use the kernel size of 11 and stride of 1. We do not use any dilation in it despite the fact that they could increase the temporal receptive field: we found that varying the kernel size didn't produce much benefit in terms of video quality. Using padding-less convolutions allows the model to be stable when unrolled at large depths. We use 2 layers of such convolutions with a hidden size of 512. Another benefit of using `conv1d`-based blocks is that in contrast to LSTM/GRU cells one can practically incorporate equalized learning rate [28] scheme into it.

Using `conv1d`-based motion mapping network without paddings forces us to use "previous" motion noise codes $\boldsymbol{z}_t^\mathrm{m}$. That's why instead of sampling a sequence $\boldsymbol{z}_{t_0}^\mathrm{m}, ..., \boldsymbol{z}_{t_n}^\mathrm{m}$, we sample a slightly larger one to adjust for the reduced sequence size. For the same-padding strategy, for sampling

[7] https://github.com/nvlabs/stylegan2-ada

a frame at position $t \in [t_{n-1}, t_n)$, we would need to produce n motion noise codes $\boldsymbol{z}^{\mathsf{m}}$. But with our kernel size of 11, with 2 layers of convolutions and without padding, the resulted sequence size is $n + 20$.

The training performance of VideoGPT on UCF101 is surprisingly low despite the fact that it was developed for such kind of datasets [80]. We hypothesize that this happens due to UCF101 being a very difficult dataset and VideoGPT being trained with the batch size of 4 (higher batch size didn't fit our 200 GB GPU memory setup), which damaged its ability to learn the distribution.

To train our model, we also utilized adaptive differentiable augmentations of StyleGAN2-ADA [29], but we found it important to make them *video-consistent*, i.e. applying the same augmentation for each frame of a video. Otherwise, the discriminator starts to underperform, and the overall quality decreases. We use the default `bgc` augmentations pipe from StyleGAN2-ADA, which includes horizontal flips, 90 degrees rotations, scaling, horizontal/vertical translations, hue/saturation/brightness/contrast changes and luma axis flipping.

While training the model, for real videos we first select a video index and then we select random clip (i.e., a clip with a random offset). This differs from the traditional DIGAN or VideoGPT training scheme, that's why we needed to change the data loaders to make them learn the same statistics and not get biased by very long videos.

To develop this project, ≈7.5 NVidia v100 32GB GPU-years + ≈0.3 NVidia A6000 GPU-years were spent.

B.2. Projection and editing procedures

In this subsection, we describe the embedding and editing procedures, which were used to obtain results in Fig 2.

Projection. To project an existing photogrpah into the latent space of G, we used a procedure from StyleGAN2 [31], but projecting into $\mathcal{W}+$ space [1] instead of $\mathcal{W}$, since it produces better reconstruction results and does not spoil editing properties. We set the initial learning rate to 0.1 and optimized a $\boldsymbol{w}$ code for LPIPS reconstruction loss [81] for 1000 steps using Adam. For motion codes, we initializated a static sequence and kept it fixed during the optimization process. We noticed that when it is also being optimized, the reconstruction becomes almost perfect, but it breaks when another sequence of motion codes is plugged in.

Editing. Our CLIP editing procedure is very similar to the one in StyleCLIP [48], with the exception that we embed an image assuming that it is a video frame in location $t = 0$. On each iteration, we resample motion codes since all our edits are semantic and do not refer to motion. We leave the motion editing with CLIP for future exploration. For the sky editing video presented in Fig 2, we additionally utilize masking: we initialize a mask to cover the trees and try not to change them during the optimization using LPIPS loss. For all the videos, presented in the supplementary website, *no masking is used*.

The details can be found in the provided source code.

B.3. Additional details on positional embeddings

Mitigating high-frequency artifacts. We noticed that if our periods $\boldsymbol{\omega}_t$ are left unbounded, they might grow to very large values (up to magnitude of ≈20.0), which corresponds to extra high frequencies (the period length becomes less than 4 frames) and leads to temporal aliasing. That's why we process them via the $\tanh(\boldsymbol{\omega}_t)+1$ transform: this bounds them into $(0, 2)$ range with the mean of 1.0, i.e. using the at-initialization frequency scaling, which we discuss next.

Linearly spaced periods. An important design decision is the scaling of periods since at initialization it should cover both high-frequency and low-frequency details. Existing works use either exponential scaling $\boldsymbol{\sigma} = (2\pi/2^d, 2\pi/2^{d-1}, ...)$ (e.g., [23, 39, 45, 62]) or random scaling $\boldsymbol{\sigma} \sim \mathcal{N}(0, \xi \boldsymbol{I})$ (e.g., [4, 59, 61, 65]). In practice, we scale the i-th column of the amplitudes weight matrix with the value:

$$\sigma_i = \frac{2\pi}{\omega_{\text{min}} + (i/N) \cdot (\omega_{\text{max}} - \omega_{\text{min}})}, \quad (6)$$

where we use $\omega_{\text{max}} = 2^{10}$ frames and $\omega_{\text{min}} = 2^3$ frames in all the experiments, except for SkyTimelapse, for each we use $\omega_{\text{min}} = 2^8$. We call this scheme *linear scaling* and use it as an additional tool to alleviate periodicity since it greatly increases the overall cycle of a positional embedding (see Fig 9). See also the accompanying source code for details.

Another benefit of using our positional embeddings over LSTM is that they are "always stable", i.e. they are always in a suitable range.

C. Evaluation details

For the practical implementation, see the provided source code: https://github.com/universome/stylegan-v.

In this section, we describe the difficulties of a fair comparison of the FVD score. There are discrepancies between papers in computing even FID [47]. So, it is less surprising that computing FVD for videos diverge even more and has even more implications for methods evaluation.

First, we note that I3D model [10] has different weights on tf.hub https://tfhub.dev/deepmind/i3d-kinetics-400/1 — the model which is used in the official FVD repo.[8] — compared to its official release in the official github repo implementation [9] That's why we manually exported the weights

[8] https://github.com/google-research/google-research/blob/master/frechet_video_distance

[9] https://github.com/deepmind/kinetics-i3d

from tf.hub and used this github repo [10] to obtain an exact implementation in Pytorch.

There are several issues with FVD metric on its own. First, it does not capture motion collapse, which can be observed by comparing FVD_{16} and FVD_{128} scores between StyleGAN-V and StyleGAN-Vwith LSTM motion codes instead of our ones: the latter one has a severe motion collapse issue (see the samples on our website) and has similar or lower FVD_{128} scores compared to our model: 196.1 or 165.8 (depending on the distance between anchors) vs 197.0 for our model. Another issue with FVD calculation is that it is biased towards image quality. If one trains a good image generator, i.e. a model which is not able to generate any videos at all, then FVD will still be good for it even despite the fact that it would have degenerate motion.

We also want to make a note on how we compute FID for vidoe generators. For this, we generate 2048 videos of 16 frames each (starting with $t = 0$) and use all those frames in the FID computation. In this way, it gives ≈33k images to construct the dataset, but those images will have lower diversity compared to a typically utilized 50k-sized set of images from a traditional image generator [30]. The reason of it is that 16 images in a single clip likely share a lot of content. A better strategy would be to generate 50k videos and pick a random frame from each video, but this is too heavy computationally for models which produce frames autoregressively. And using just the first frame in FID computation will unfairly favour MoCoGAN-HD, which generates the very first frame of each video with a freezed StyleGAN2 model.

FVD is greatly influenced by 1) how many clips per video are selected; 2) with which offsets; and 3) at which frame-rate. For example, SkyTimelapse contains several *extremely* long videos: if we select as many clips as possible from each real video, that it will severely bias the statistics of FVD. For FaceForensics, videos often contain intro frames during their first ≈0.5-1.0 seconds, which will affect FVD when a constant offset of 0 is chosen to extract a single clip per video.

That's why we use the following protocol to compute FVD_n.

Computing real statistics. To compute real statistics, we select a *single* clip per video, chosen at a random offset. We use the actual frame-rate of the dataset, which the model is being trained on, without skipping any frames. The problem of such an approach is that for datasets with small number of long videos (like, FaceForensics, see Table 7) might have noisy estimates. But our results showed that the standard deviations are always < 3.0 even for FaceForensics 256^2. The largest standard deviation we obserbed was when computing FVD_{16} on RainbowJelly: on this dataset it was 26.15 for VideoGPT, but it is $< 1\%$ of its overall magnitude.

[10] https://github.com/hassony2/kinetics_i3d_pytorch

Table 4. Subtleties of FVD calculation. We report different ways of calculating FVD_{16} on FaceForensics 256^2 (FF) and SktTimelapse 256^2 (ST) for one of our checkpoints. We show how the scores of StyleGAN-V are influenced a lot when different strategies of FVD_{16} calculation are employed. See the text for the description of each row.

Method	FF	ST
Proper computation	76.82 ±1.57	61.95 ±0.92
When resized to 128^2	38.92	59.86
With jpg/png discrepancy	80.17	71.40
When using all clips per video	84.59	72.03
When using only first frames	91.64	59.74
When using subsampling of $s = 8$	82.88	90.21
Still real images	342.5	166.8

Computing fake statistics. To compute fake statistics, we generate 2048 videos and save them as frames in JPEG format via the Pillow library. We use the quality parameter $q = 95$ for doing this, since it was shown to have very close quality to PNG, but without introducing artifacts that would lead to discrepancies [47]. Ideally, one would like to store frames in the PNG format, but in this case it would be too expensive to represent video datasets: for example, MEAD 1024^2 would occupy ≈0.5 terabytes of space in this case.

We illustrate the subtleties of FVD computation in Table 4. For this, we compute real/fake statistics for our model in several different ways:

- *Resized to* 128^2. Both fake and real statistics images are resized into 128^2 resolution via the pytorch bilinear interpolation (without corners alignment) before computing FVD.

- *JPG/PNG discrepancy*. Instead of saving fake frames in JPG with $q = 95$, we use $q = 75$ parameter in the PIL library. This creates more JPEG-like artifacts, which, for example, FID is very sensitive to.

- *Using all clips per video*. We use all available n-frames-long clips in each video without overlaps. Note, that our model was trained

- *Using only first frames*. In each real video, instead of using random offsets to select clips, we use the first n frames.

- *Using $s = 8$ subsampling*. When sampling frames for computing real/fake statistics, we select each 8-th frame. This is the strategy which was employed for some of the experiments in the original paper [69] — but in their case, authors trained the model on videos with this subsampling.

Table 5. Inception Score [55] on UCF101 256^2 (note that the underlying C3D model resizes the 256^2 videos into 112^2 resolution under the hood, eliminating high-quality details).

Method	Inception Score [55]
MoCoGAN [68]	10.09±0.30
MoCoGAN+SG2 (ours)	15.26±0.95
VideoGPT [80]	12.61±0.33
MoCoGAN-HD [66]	23.39±1.48
DIGAN [5]	23.16±1.13
StyleGAN-V (ours)	23.94±0.73
Real videos	97.23±0.38

Table 6. FVD_{16}, FID and training costs of modern video generators on FaceForensics 256^2. Training cost is measured in terms of GPU-days.

Method	FVD_{16}	FID	Training cost
MoCoGAN [68]	124.7	23.97	5
MoCoGAN+SG2 (ours)	55.62	10.82	8
VideoGPT [80]	185.9	22.7	32
MoCoGAN-HD [66]	111.8	7.12	16.5
DIGAN [5]	62.5	19.1	16
StyleGAN-V (ours)	47.41	9.445	8
StyleGAN2 [29]	N/A	8.42	7.72

For completeness, we also provide the Inception Score [55] on UCF-101 256^2 dataset in Table 5. Note that is computed by resizing all videos to 112×112 spatial resolution (due to the internal structure of the C3D [67] model), which makes it impossible for it to capture high-resolution details of the generated videos, which is the focus of the current work.

In Tab 6, we provide the numbers, used in Fig 3. Note that StyleGAN2 training in our case is slightly slower than the officially specified one (7.3 vs 7.7 GPU days)[11], which we attribute to a slightly slower file system on our computational cluster.

D. Failed experiments

In this section, we provide a list of ideas, which we tried to make work, but they didn't work either because the idea itself is not good, or because we didn't put enough experimental effort into investigating it.

Hierarchical motion codes. We tried having several layers of motion codes. Each layer has its own distance between the codes. In this way, high-level codes should capture high-level motion and bottom-level codes should represent short local motion patterns. This didn't improve the scores and didn't provide any disentanglement of motion information. We believe that the motion should be represented differently (similar to FOMM [58]), rather than with motion codes, because they make it difficult for G to make them temporily coherent.

Maximizing entropy of motion codes to alleviate motion collapse. As an additional tool to alleviate motion collapse, we tried to maximize entropy of wave parameters of our motion codes. The generator solved the task of maximizing the entropy well, but it didn't affect the motion collapse: it managed to save some coordination dimensions of $\boldsymbol{v}_t$ specifically to synchronize motions.

Prorgressive growing of frequences in positional embeddings. We tried starting with low-frequencies first and progressively open new and new ones during the training. It is a popular strategy for training implicit neural representations on reconstruction tasks (e.g., [23, 45]), but in our case we found the following problem with it. The generator learned to use low frequencies for representing high-frequency motion and didn't learn to utilize high frequencies for this task when they became available. That's why high-frequency motion patterns (like blinking or speaking) were unnaturally slow.

Continuous LSTM with EMA states. Our motion codes use sine/cosine activations, which makes them suffer from periodic artifacts (those artifacts are mitigated by our parametrization, but still present sometimes). We tried to use LSTM, but with *exponential moving average* on top of its hidden states to smoothen out motion representations temporally. However, (likely due to the lack of experimental effort which we invested into this direction), the resulted motion representations were either too smooth or too sharp (depending on the EMA window size), which resulted in unnatural motions.

Concatenating spatial coordinates. INR-GAN [61] uses spatial positional embeddings and shows that they provide better geometric prior to the model. We tried to use them as well in our experiments, but they didn't provide any improvement neither in qualitatively, nor quantitatively, but made the training slightly slower (by $\approx$%10) due to the increased channel dimensionalities.

Feature differences in D. Another experiment direction which we tried is computing differences between activations of next/previous frames in a video and concatenating this information back to the activations tensor. The intuition was to provide D information with some sort of "latent" optical flow information. However, it made D too powerful (its loss became smaller than usual) and it started to outpace G too much, which decreased the final scores.

Predicting δ^x instead of conditioning in D. There are two ways to utilize the time information in D: as a conditioning signal or as a learning signal. For the latter one, we tried to predict the time distances between frames by train-

[11] https://github.com/NVlabs/stylegan2-ada-pytorch

Table 7. Additional datasets information in terms of total lengths (in the total number of hours), average video length (in seconds), frame rate and the amount of speakers (for FaceForensics and MEAD).

Dataset	#hours	avg len	FPS	#speakers
FaceForensics [54]	4.04	20.7s	25	704
SkyTimelapse [79]	12.99	22.1s	25	N/A
UCF-101 [63]	0.51	6.8s	25	N/A
RainbowJelly	7.99	17.1s	30	N/A
MEAD [73]	36.11	4.3s	30	48

ing an additional head to predict the class (we treated the problem as classification instead of regression since there is a very limited amount of time distances between frames which D sees during its training). However, it noticeably decreased the scores.

Conditioning on video length. For *unconditional* UCF-101, it might be very important for G to know the video length in advance. Because some classes might contain very short clips (like, jumping), while others are very long, and it might be useful for G to know in advance which video it will need to generate (since we sample frames at random time locations during training). However, utilizing this conditioning didn't influence the scores.

E. Datasets details

E.1. Datasets details

We provide the dataset statistics in Fig 10 and their comparison in Table 7. Note, that for MEAD, we use only its front camera shots (originally, it releases shots from several camera positions).

E.2. Rainbow Jelly

We noticed that modern video synthesis datasets are either too simple or too difficult in terms of content and motion, and there are no datasets "in-between". That's why we introduce RainbowJelly: a dataset of "floating" jellyfish. It is constructed from an 8-hour-long movie in 4K resolution and 30 FPS from the Hoccori Japan youtube video channel. It contains simple content but complex hierarchical motions and this makes it a challenging but approachable test-bed for evaluating modern video generators.

For our RainbowJelly benchmark, we used the following film: https://www.youtube.com/watch?v=P8Bit37hlsQ. We cannot release this dataset due to the copyright restrictions, but we released a full script which processes it (see the provided source code). To construct a benchmark, we sliced it into 1686 chunks of 512 frames each, starting with the 150-th frame (to remove the loading screen), center-cropped and resized into 256^2 resolution. This benchmark is advantageous compared to the existing ones in the following way:

1. It contains complex hierarchical motions:
    - a jellyfish flowing in a particular direction (low-frequency global motion);
    - a jellyfish pushing water with its arms (medium-frequency motion)
    - small perturbations of jellyfish's body and tentacles (high-frequency local motion).
2. It is a very high-quality dataset (4K resolution).
3. It is simple in terms of content, which makes the benchmark more focused on motions.
4. It contains long videos.

F. Implicit assumptions of sparse training

In this section, we elaborate on our simple theoretical exposition from §3.3

Consider that we want to fit a probabilistic model $q_\theta(\boldsymbol{x})$ to the real data distribution $\boldsymbol{x} \sim p(\boldsymbol{x}) = p(x_1, ..., x_n)$. For simplicity, we will be considering a discrete finite case, i.e. $n < \infty$, but note that videos, while continuous and infinite in theory, are still discretized and have a time limit to fit on a computer in practice. For fitting the distribution, we use k*-sparse training*, i.e. picking only k random coordinates from each sample $\boldsymbol{x} \sim p(\boldsymbol{x})$ during the optimization process. In other words, introducing k-sparse sampling reformulates the problem from

$$d(p(\boldsymbol{x}), q_\theta(\boldsymbol{x})) \longrightarrow \min_\theta \tag{7}$$

into

$$\sum_{I \in \mathcal{I}^k} d(p(\boldsymbol{x}_I), q_\theta(\boldsymbol{x}_I)) \longrightarrow \min_\theta, \tag{8}$$

where $d(\cdot,\cdot)$ is a problem-specific distance function between probability distributions, $\mathcal{I}^k$ is a collection of all possible sets $I = \{i_1, ..., i_k\}$ of unique indices $i_j \in \{1, 2, ..., n\}$ and $\boldsymbol{x}_I$ denotes a sub-vector $(x_{i_1}, ..., x_{i_k})$ of $\boldsymbol{x}$. This means, that instead of bridging together full distributions we choose to bridge all their possible marginals of length k instead. When solving Eq. (8) will help us to obtain the full joint distribution $p(\boldsymbol{x})$? To investigate this question, we develop the following simple statement.

Let's denote by $\mathcal{J}^k_{<i}$ a collection of sets J_i of up to k indices s.t. $\forall J_i \in \mathcal{J}^k_{<i}$ we have $j < i$ for all $j \in J_i$.

Using the chain rule, we can represent $p(\boldsymbol{x})$ as:

$$p(\boldsymbol{x}) = \prod_{i=1}^n p(x_i | \boldsymbol{x}_{<i}), \tag{9}$$

where $\boldsymbol{x}_{<i}$ denotes the sequence $(x_1, ..., x_{i-1})$. Now, if we know that for each i, there exists $J_i = \{j_1, ..., j_{k-1}\}$ with $j_\ell < i$ s.t.:

$$p(x_i | \boldsymbol{x}_{<i}) = p(x_i | \boldsymbol{x}_{J_i}), \tag{10}$$

then $p(\boldsymbol{x})$ is obviously simplified to:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{<i}) = \prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{J_i}) \tag{11}$$

Does this tell anything useful? Surprisingly, yes. It says that if $p(\boldsymbol{x})$ is simple enough that instead of using the whole history $\boldsymbol{x}_{<i}$ to model $p(x_i|\boldsymbol{x}_{<i})$ it's enough to use only some set "representative moments" J_i (unique for each i) with the size $|J_i| < k$, then k-sparse training is a viable alternative. After fitting $q_\theta(\boldsymbol{x})$ via k-sparse training, we will be able to obtain $p(\boldsymbol{x})$ using Eq (10) *even though* $q_\theta(\boldsymbol{x}) \not\equiv p(\boldsymbol{x})$! Note, that one can obtain a conditional distributional $p(x_i|\boldsymbol{x}_I)$ from the marginal one $p(x_i, \boldsymbol{x}_I)$ for some set of indicies $I = \{i_1, ..., i_{\ell-1}\}$ via:

$$p(x_i|\boldsymbol{x}_I) = \frac{p(x_i, \boldsymbol{x}_I)}{p(\boldsymbol{x}_I)} = \frac{p(x_i, \boldsymbol{x}_I)}{\int_{x_i} p(x_i, \boldsymbol{x}_I) dx_i}. \tag{12}$$

But we would also like to have the "reverse" dependency, i.e. knowing that if we can approximate the distribution via a set of marginals, then this distribution is not too difficult. For this claim, we will need to consider marginals not of an arbitrary form $p(\boldsymbol{x}_S)$, but of the form $p(x_i, J_i)$, and we would need exactly n of those. The reverse implication is the following. *If $p(\boldsymbol{x})$ can be represented as a product of n conditionals $p(i|J_i)$, then for each i there exists $J_i \in \mathcal{J}^k_{<i}$ s.t. $p(x_i|\boldsymbol{x}_i) = p(x_i|J_i)$.* This statement, just like the previous one, looks obvious. But oddly, requires more than a single sentence to prove. First, we are given that:

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{<i}) = \prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{J_i}), \tag{13}$$

but unfortunately, we cannot directly claim that each term in the product $\prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{<i})$ equals to its corresponding one in the product $\prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{J_i})$. For this, we first need to show that for each m we have:

$$p(\boldsymbol{x}_{\leq m}) = \prod_{i=1}^{m} p(x_i|\boldsymbol{x}_{J_i}) \tag{14}$$

It can be seen from the fact, that:

$$\begin{aligned} p(\boldsymbol{x}_{\leq m}) &= \int_{\boldsymbol{x}_{>m}} p(\boldsymbol{x}) d\boldsymbol{x}_{>m} \\ &= \int_{\boldsymbol{x}_{>m}} \prod_{i=1}^{n} p(x_i|\boldsymbol{x}_{J_i}) d\boldsymbol{x}_{>m} \\ &= \prod_{i=1}^{m} p(x_i|\boldsymbol{x}_{J_i}) \cdot \int_{\boldsymbol{x}_{>m}} \prod_{i=m+1}^{n} p(x_i|\boldsymbol{x}_{J_i}) d\boldsymbol{x}_{>m} \\ &= \prod_{i=1}^{m} p(x_i|\boldsymbol{x}_{J_i}) \cdot 1 \\ &= \prod_{i=1}^{m} p(x_i|\boldsymbol{x}_{J_i}) \end{aligned} \tag{15}$$

This allows to cancel terms in the chain rule one by one, starting from the end, leading to the desired equality:

$$p(x_i|\boldsymbol{x}_{<i}) = p(x_i|\boldsymbol{x}_{J_i}) \tag{16}$$

Does this reverse claim tells us anything useful? Surprisingly again, yes. It implies that if we managed to fit $p(\boldsymbol{x})$ by using k-sparse training, then this distribution is not sophisticated.

Merging the above two statements together, we see that *$p(\boldsymbol{x})$ can be represented as a product of n conditionals $p(x_i|\boldsymbol{x}_{J_i})$ for $i = 1, ..., n$ if and only if for all $i \leqslant n$ there exists $J_i \in \mathcal{J}^{k-1}_{<i}$ s.t. $p(x_i|\boldsymbol{x}_{<i}) \equiv p(x_i|\boldsymbol{x}_{J_i})$.*

What does this statement tell for video synthesis? Any video synthesis algorithm utilizes k-sparse training to learn its underlying model, but in contrast to prior work, we use very small values of k. This means, that we fit our model $q_\theta(\boldsymbol{x})$ to model any k-marginals of $p(\boldsymbol{x})$ (considering that we pick frames uniformly at random) instead of the full one $p(\boldsymbol{x})$. And using the above statement, such a setup implies the assumption of Eq (10). This equation says that one can know everything about x_i by just observing previous frames J_i. In other words, x_i must be predictable from $\boldsymbol{x}_{J_i}$. Moreover, it is easy to show that our statement can generalize to include several $J_i^{(1)}, ..., J_i^{(\ell)}$ for i-th frame, i.e. there might exist several explainable sets of frames.

G. Additional samples

For the ease of visualization, we provide additional samples of the model via a web page: https://universome.github.io/stylegan-v.

H. Comparison to DIGAN

Our model shares a lot of similarities to DIGAN [5] and in this section we highlight those similarities and differences.

H.1. Major similarities

Sparse training. DIGAN also utilizes very sparse training (only 2 frames per video). But in our case, we additionally explore the optimal number of frames per video k (see §3.3).

Continuous-time generator. DIGAN also builds a generator, which is continuous in time. But our generator does not lose the quality at infinitely large lengths.

Dropping `conv3d` blocks. DIGAN also drops `conv3d` blocks in their discriminator. But in contrast to us, they still have 2 discriminators.

H.2. Major differences

Motion representation. DIGAN uses only a single global motion code, which makes it *theoretically* impossible to generate infinite videos: at some point it will start repeating itself (due to the usage of sine/cosine-based positional embeddings). In our case, we use an infinite sequence of motion codes, which are being temporally interpolated, computed wave parameters from and transformed into motion codes. DIGAN mixes temporal and spatial information together into the same positional embedding, which creates the following problem: even when time changes, the spatial location, perceived by the model, *also changes*. This creates a "head-flying-away" effect (see the samples). In our case, we keep these two information sources decomposed from one another.

Generator's backbone. DIGAN is built on top of INR-GAN [61], while our work uses StyleGAN2. This allows DIGAN to inherit INR-GAN's benefits from being spatially continuous, but at the expense of being less stable and being slower to train (due to the lack of mixed precision and increased channel dimensionalities from concatenating positional embeddings).

Discriminator structure. DIGAN uses *two* discriminators: the first one operates on image-level and is equivalent to StyleGAN2's one, while the other one operates on "video" level and takes frames $\boldsymbol{x}_{t_1}, \boldsymbol{x}_{t_2}$ and the time differences between them $\Delta = t_2 - t_1$, concatenates them all together into a 7-channel input image (tiling the time difference scalar) and passes into a model with StyleGAN2 discriminator's backbone. In our case, we use concatenate the frames features and apply the conditioning via the projection discriminator [40] strategy.

Sampling procedure. We use $k = 3$ samples per video, while DIGAN uses $k = 2$. Also, we sample frames uniformly randomly, while DIGAN selects $t_1 \sim \text{Beta}(2, 1)$ and $t_1 \sim \text{Beta}(1, 2)$ (in this way, DIGAN sometimes have $t_1 > t_2$). Apart from that, they use $T = 16$.

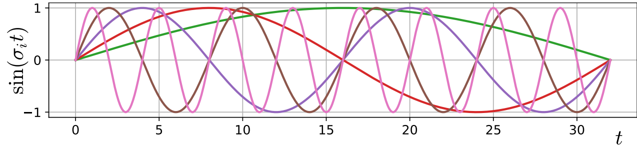
Apart from those major distinctions, there are lot of small implementation differences. We refer an interested reader to the released codebases for them:

- StyleGAN-V: https://github.com/universome/stylegan-v
- DIGAN: https://openreview.net/forum?id=Czsdv-S4-w9

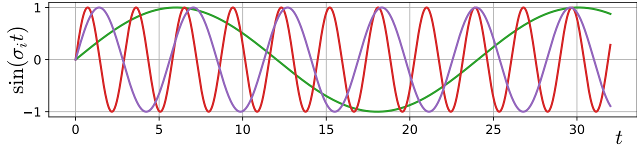
H.3. A note on the computational cost

INR-GAN demonstrated that it has higher throughput than StyleGAN2 in terms of images/second [61]. But the authors compare to the original StyleGAN2 implementation and not to the one from StyleGAN2-ADA repo, which is *much* better optimized. Also, they use caching of positional embeddings which is only possible at test-time and has great influence on its computational performance. In this way, we found that that StyleGAN2 is $\approx$2 times faster to train and is *less* consuming in terms of GPU memory than INR-GAN.

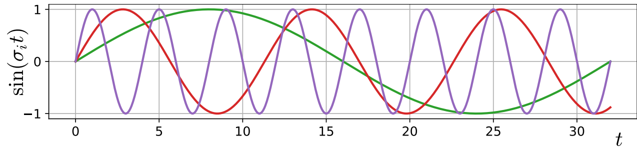
DIGAN is based on top of INR-GAN and that's why suffers from the issues described above. We trained it for a week on $\times 4$ v100 NVidia GPUs and observed that it stopped improving after $\approx$5 days of training. This is equivalent to $\approx 20k$ real frames seen by the discriminator (while MoCoGAN+SG2 and StyleGAN-V reach $\approx 25k$ in just 2 days for the same resolution in the same environment). For the time of the submitting the main paper, there was no information about the training cost. However, the authors updated their manuscript for the time of submitting the supplementary and specify the training cost of 8 GPU-days 128^2 *resolution*, which is consistent with our experiments (considering that we have twice as larger resolution).



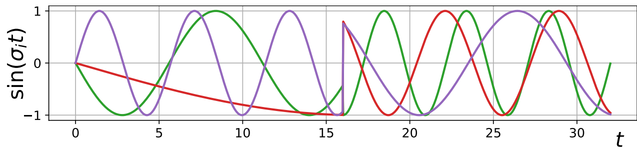
(a) Exponentially spaced periods [39]: $d = 5$, cycle length is 64.



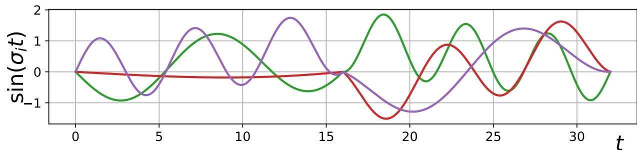
(b) Random periods [59, 65]: $d = 3$, cycle length is 120 (for the depicted $\boldsymbol{\sigma} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$).



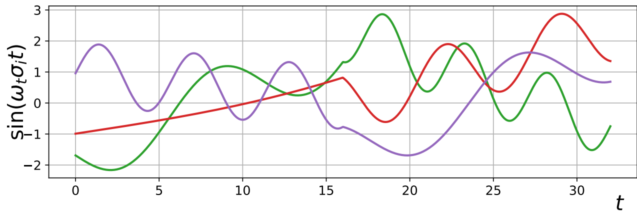
(c) Linearly spaced periods (ours): $d = 3$, cycle length is 352.



(d) Raw acyclic positional embeddings $\tilde{\boldsymbol{v}}_t$: $d = 3$, no cyclicity. While such embeddings are acyclic, they have discontinuities at stitching points.

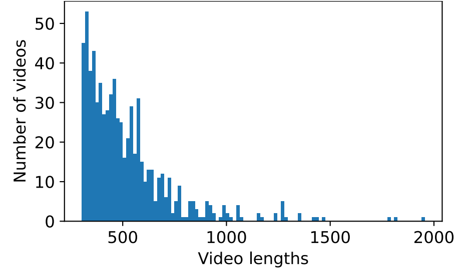


(e) Stitched raw acyclic positional embeddings without alignment vectors: $d = 3$, no cyclicity. Stitching raw positional embeddings without using "aligners" $\boldsymbol{a} = W_a \boldsymbol{u}$ removes discontinuities, but reduces the expressive power of positional embeddings since they have zero values at time locations $\{t_0, t_1, ..., t_n, ...\}$.

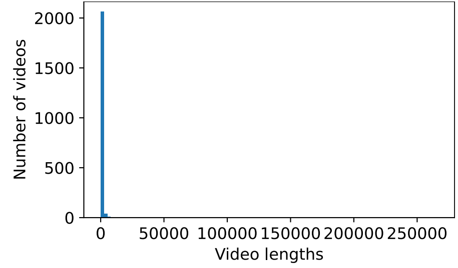


(f) Acyclic periods with linearly-spaced scaling (ours): $d = 3$, no cyclicity. Notice that the frequencies and phases are controlled by the motion mapping network F_m: for example, it has the possibility to accelerate some motion (like the one represented by the red curve) by increasing its frequency.

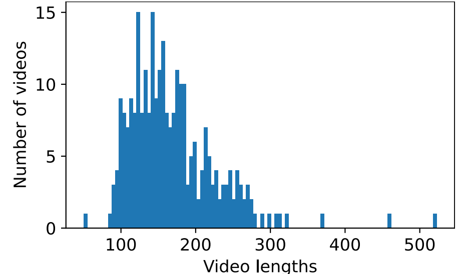
Figure 9. Visualizing positional embeddings $\sin(\boldsymbol{\sigma} t)$ for different initialization strategies of periods scales $\boldsymbol{\sigma}$. The cycle length is the minimum value of t for which the positional embedding vector starts repeating itself (it is computed as a least common multiple of all the individual periods lengths). Existing works use either exponentially spaced or random scaling, but in our case we use the linearly spaced one since it has a very large global cycle (in contrast to exponential scaling) and is guaranteed to include high-frequency, medium-frequency and high-freqency waves (in contrast to random scaling).



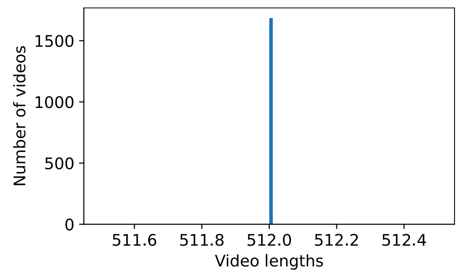
(a) FaceForensics [54].



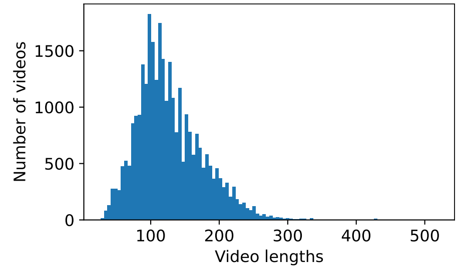
(b) SkyTimelapse [79].



(c) UCF-101 [63].



(d) RainbowJelly [63].



(e) MEAD [63].

Figure 10. Distribution of video lengths (in terms of numbers of frames) for different datasets. Note that RainbowJelly and MEAD [73] are 30 FPS, while the rest are 25 FPS datasets. Note that SkyTimelapse contains several very long videos which might bias the distribution if not treated properly.