tempoGAN: A Temporally Coherent, Volumetric GAN for Super-resolution Fluid Flow

YOU XIE*, Technical University of Munich
ERIK FRANZ*, Technical University of Munich
MENGYU CHU*, Technical University of Munich
NILS THUEREY, Technical University of Munich

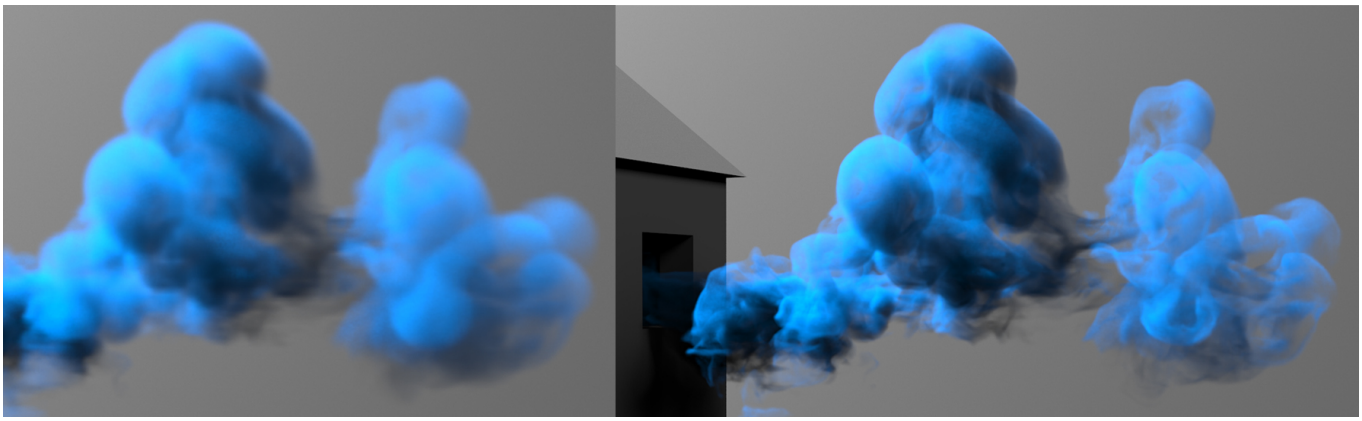

Fig. 1. Our convolutional neural network learns to generate highly detailed, and temporally coherent features based on a low-resolution field containing a single time-step of density and velocity data. We introduce a novel discriminator that ensures the synthesized details change smoothly over time.

We propose a temporally coherent generative model addressing the super-resolution problem for fluid flows. Our work represents a first approach to synthesize four-dimensional physics fields with neural networks. Based on a conditional generative adversarial network that is designed for the inference of three-dimensional volumetric data, our model generates consistent and detailed results by using a novel temporal discriminator, in addition to the commonly used spatial one. Our experiments show that the generator is able to infer more realistic high-resolution details by using additional physical quantities, such as low-resolution velocities or vorticities. Besides improvements in the training process and in the generated outputs, these inputs offer means for artistic control as well. We additionally employ a physics-aware data augmentation step, which is crucial to avoid overfitting and to reduce memory requirements. In this way, our network learns to generate advected quantities with highly detailed, realistic, and temporally coherent features. Our method works instantaneously, using only a single time-step of low-resolution fluid data. We demonstrate the abilities of our method using a variety of complex inputs and applications in two and three dimensions.

CCS Concepts: • **Computing methodologies** → **Neural networks**; **Physical simulation**;

Additional Key Words and Phrases: physics-based deep learning, generative models, computer animation, fluid simulation

ACM Reference Format:
You Xie*, Erik Franz*, Mengyu Chu*, and Nils Thuerey. 2018. tempoGAN: A Temporally Coherent, Volumetric GAN for Super-resolution Fluid Flow. *ACM Trans. Graph.* 37, 4, Article 95 (August 2018), 15 pages. https://doi.org/10.1145/3197517.3201304

1 INTRODUCTION

Generative models were highly successful in the last years to represent and synthesize complex natural images [Goodfellow et al. 2014]. These works demonstrated that deep convolutional neural networks (CNNs) are able to capture the distribution of, e.g., photos of human faces, and generate novel, previously unseen versions that are virtually indistinguishable from the original inputs. Likewise, similar algorithms were shown to be extremely successful at generating natural high-resolution images from a coarse input [Karras et al. 2017]. However, in their original form, these generative models do not take into account the temporal evolution of the data, which is crucial for realistic physical systems. In the following, we will extend these methods to generate high-resolution volumetric data sets of passively advected flow quantities, and ensuring temporal coherence is one of the core aspects that we will focus on below. We

(*) Similar amount of contributions.
Authors' addresses: You Xie*, Technical University of Munich, you.xie@tum.de; Erik Franz*, Technical University of Munich, franzer@in.tum.de; Mengyu Chu*, Technical University of Munich, mengyu.chu@tum.de; Nils Thuerey, Technical University of Munich, nils.thuerey@tum.de.

0730-0301/2018/8-ART95 $15.00
https://doi.org/10.1145/3197517.3201304

will demonstrate that it is especially important to make the training process aware of the underlying transport phenomena, such that the network can learn to generate stable and highly detailed solutions.

Capturing the intricate details of turbulent flows has been a long-standing challenge for numerical simulations. Resolving such details with discretized models induces enormous computational costs and quickly becomes infeasible for flows on human space and time scales. While algorithms to increase the apparent resolution of simulations can alleviate this problem [Kim et al. 2008], they are typically based on procedural models that are only loosely inspired by the underlying physics. In contrast to all previous methods, our algorithm represents a physically-based interpolation, that does not require any form of additional temporal data or quantities tracked over time. The super-resolution process is instantaneous, based on volumetric data from a single frame of a fluid simulation. We found that inference of high-resolution data in a fluid flow setting benefits from the availability of information about the flow. In our case, this takes the shape of additional physical variables such as velocity and vorticity as inputs, which in turn yield means for artistic control. A particular challenge in the field of super-resolution flow is how to evaluate the quality of the generated output. As we are typically targeting turbulent motions, a single coarse approximation can be associated with a large variety of significantly different high-resolution versions. As long as the output matches the correlated spatial and temporal distributions of the reference data, it represents a correct solution. To encode this requirement in the training process of a neural network, we employ so-called generative adversarial networks (GANs). These methods train a *generator*, as well as a second network, the *discriminator* that learns to judge how closely the generated output matches the ground truth data. In this way, we train a specialized, data-driven loss function alongside the generative network, while making sure it is differentiable and compatible with the training process. We not only employ this adversarial approach for the smoke density outputs, but we also train a specialized and novel adversarial loss function that learns to judge the temporal coherence of the outputs.

We additionally present best practices to set up a training pipeline for physics-based GANs. E.g., we found it particularly useful to have physics-aware data augmentation functionality in place. The large amounts of space-time data that arise in the context of many physics problems quickly bring typical hardware environments to their limits. As such, we found data augmentation crucial to avoid overfitting. We also explored a variety of different variants for setting up the networks as well as training them, and we will evaluate them in terms of their capabilities to learn high-resolution physics functions below.

To summarize, the main contributions of our work are:

- a novel temporal discriminator, to generate consistent and highly detailed results over time,
- artistic control of the outputs, in the form of additional loss terms and an intentional entangling of the physical quantities used as inputs,
- a physics aware data augmentation method,
- and a thorough evaluation of adversarial training processes for physics functions.

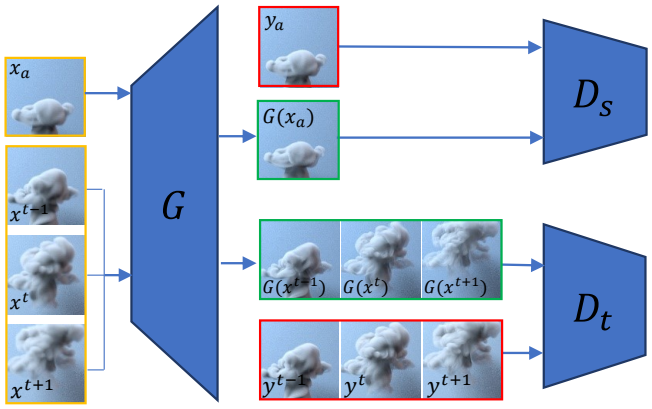

Fig. 2. This figure gives a high level overview of our approach: a generator on the left, is guided during training by two discriminator networks (right), one of which focuses on space (D_s), while the other one focuses on temporal aspects (D_t). At runtime, both are discarded, and only the generator network is evaluated.

To the best of our knowledge, our approach is the first generative adversarial network for four-dimensional functions, and we will demonstrate that it successfully learns to infer solutions for flow transport processes from approximate solutions. A high level preview of the architecture we propose can be found in Fig. 2.

2 RELATED WORK

In the area of computer vision, deep learning techniques have achieved significant breakthroughs in numerous fields such as classification [Krizhevsky et al. 2012], object detection [Girshick et al. 2014], style transfer [Luan et al. 2017], novel view synthesis [Flynn et al. 2016], and additionally, in the area of content creation. For more in-depth reviews of neural networks and deep learning techniques, we refer the readers to corresponding books [Bishop 2006; Goodfellow et al. 2016].

One of the popular methods to generate content are so called *generative adversarial networks* (GANs), introduced by Goodfellow et al. [Goodfellow et al. 2014]. They were shown to be particularly powerful at re-creating the distributions of complex data sets such as images of human faces. Depending on the kind of input data they take, GANs can be separated into unconditional and conditional ones. The formers generate realistic data from samples of a synthetic data distribution like Gaussian noise. The DC-GAN [Radford et al. 2016] is a good example of an unconditional GAN. It was designed for generic natural images, while the cycle-consistent GAN by Zhu et al. [2017] was developed to translate between different classes of images. The conditional GANs were introduced by Mirza and Osindero [2014], and provide the network with an input that is in some way related to the target function in order to control the generated output. Therefore, conditional variants are popular for transformation tasks, such as image translations problems [Isola et al. 2017] and super resolution problems [Ledig et al. 2016].

In the field of super-resolution techniques, researchers have explored different network architectures. E.g., convolutional networks

[Dong et al. 2016] were shown to be more effective than fully connected architectures. These networks can be trained with smaller tiles and later on be applied to images of arbitrary sizes [Isola et al. 2017]. Batch normalization [Lim et al. 2017] significantly improves results by removing value shifting in hidden layers, and networks employing so-called *residual blocks* [Kim et al. 2016; Lim et al. 2017] enable the training of deep networks without strongly vanishing gradients.

In term of loss functions, pixel-wise loss between the network output and the ground truth data used to be common, such as the L_1 and L_2 loss [Dong et al. 2016]. Nowadays, using the adversarial structure of GANs, or using pre-trained networks, such as the VGG net [Simonyan and Zisserman 2014] often leads to higher perceptual qualities [Johnson et al. 2016; Mathieu et al. 2015].

Our method likewise uses residual blocks in conjunction with a conditional GAN architecture to infer three-dimensional flow quantities. Here, we try to use standard architectures. While our generator is similar to approaches for image super-resolution [Ledig et al. 2016], we show that loss terms and discriminators are crucial for high-quality outputs. We also employ a fairly traditional GAN training, instead of recently proposed alternatives [Arjovsky et al. 2017; Berthelot et al. 2017], which could potentially lead to additional gains in quality. Besides the super-resolution task, our work differs from many works in the GAN area with its focus on temporal coherence, as we will demonstrate in more detail later on.

While most works have focused on single images, several papers have addressed temporal changes of data sets. One way to solve this problem is by directly incorporating the time axis, i.e., by using sequences of data as input and output. E.g., Saito et al. propose a temporal generator in their work [Saito et al. 2017], while Yu et al. [Yu et al. 2017] proposed a sequence generator that learns a stochastic policy. In these works, results need to be generated sequentially, while our algorithm processes individual frames independently, and in arbitrary order, if necessary. In addition, such approaches would explode in terms of weights and computational resources for typical four-dimensional fluid data sets.

An alternative here is to generate single frame data with additional loss terms to keep the results coherent over time. Bhattacharjee etc. [2017] achieved improved coherence in their results for video frame prediction, by adding specially designed distance measures as a discontinuity penalty between nearby frames. For video style transfer, a L_2 loss on warped nearby frames helped to alleviate temporal discontinuities, as shown by Ruder et al. [2016]. In addition to a L_2 loss on nearby frames, Chen et al. [2017] used neural networks to learn frame warping and frame combination in VGG feature space. Similarly, Liu etc. [Liu et al. 2017] used neural networks to learn spatial alignment for low-resolution inputs, and adaptive aggregation for high-resolution outputs, which also improved the temporal coherence. Due to the three-dimensional data sets we are facing, we also adopt the single frame view. However, in contrast to all previous works, we propose the use of a temporal discriminator. We will show that relying on data-driven, learned loss functions in the form of a discriminator helps to improve results over manually designed losses. Once our networks are trained, this discriminator can be discarded. Thus, unlike, e.g., aggregation methods, our approach does not influence runtime performance. While previous work shows that warping layers are useful in motion field learning [Chen et al. 2017; de Bezenac et al. 2017], our work targets the opposite direction: by providing our networks with velocities, warping layers can likewise improve the training of temporally coherent content generation.

More recently, deep learning algorithms have begun to influence computer graphics algorithms. E.g., they were successfully used for efficient and noise-free renderings [Bako et al. 2017; Chaitanya et al. 2017], the illumination of volumes [Kallweit et al. 2017], for modeling porous media [Mosser et al. 2017], and for character control [Peng et al. 2017]. First works also exist that target numerical simulations. E.g., a conditional GAN was used to compute solutions for smaller, two-dimensional advection-diffusion problems [Farimani et al. 2017; Long et al. 2017]. Others have demonstrated the inference of SPH forces with regression forests [Ladicky et al. 2015], proposed CNNs for fast pressure projections [Tompson et al. 2016], learned space-time deformations for interactive liquids [Prantl et al. 2017] , and modeled splash statistics with NNs [Um et al. 2017]. Closer to our line of work, Chu et al. [2017] proposed a method to look up pre-computed patches using CNN-based descriptors. Despite a similar goal, their methods still require additional Lagrangian tracking information, while our method does not require any modifications of a basic solver. In addition, our method does not use any stored data at runtime apart from the trained generator model.

As our method focuses on the conditional inference of high-resolution flow data sets, i.e. solutions of the *Navier-Stokes* (NS) equations, we also give a brief overview of the related work here, with a particular focus on single-phase flows. After the introduction of the stable fluids algorithm [Stam 1999], a variety of extensions and variants have been developed over the years. E.g., more accurate Eulerian advection schemes [Kim et al. 2005; Selle et al. 2008] are often employed, an alternative to which are Lagrangian versions [Magnus et al. 2011; Rasmussen et al. 2003]. While grids are more commonly used, particles can achieve non-dissipative results for which a Eulerian grid would require a significant amount of refinement. Furthermore, procedural turbulence methods to increase apparent resolutions are popular extensions [Kim et al. 2008; Narain et al. 2008; Schechter and Bridson 2008]. In contrast to our work, the different advection schemes and procedural turbulence methods require a calculation of the high-resolution transport of the density field over the full simulation sequence. Additionally, Eulerian and Lagrangian representations can be advected in a parallelized fashion on a per frame basis, in line with the application of convolutions for NNs. Our method infers an instantaneous solution to the underlying advection problem based only on a single snapshot of data, without having to compute a series of previous time steps.

Our work also shares similarities in terms of goals with other physics-based up-sampling algorithms [Kavan et al. 2011], and due to this goal, is related to fluid control methods [McNamara et al. 2004; Pan et al. 2013]. These methods would work very well in conjunction with our approach, in order to generate a coarse input with the right shape and timing.

3 ADVERSARIAL LOSS FUNCTIONS

Based on a set of low-resolution inputs, with corresponding high-resolution references, our goal is to train a CNN that produces a temporally coherent, high-resolution solution with adversarial training. We will first very briefly summarize the basics of adversarial training, and then explain our extensions for temporal coherence and for results control.

3.1 Generative Adversarial Networks

GANs consist of two models, which are trained in conjunction: the generator G and the discriminator D. Both will be realized as convolutional neural networks in our case. For regular ones, i.e., not conditional GANs, the goal is to train a generator $G(x)$ that maps a simple data distribution, typically noise, x to a complex desired output y, e.g., natural images. Instead of using a manually specified loss term to train the generator, another NN, the discriminator, is used as complex, learned loss function [Goodfellow et al. 2014]. This discriminator takes the form of a simple binary classifier, which is trained in a supervised manner to reject *generated* data, i.e., it should return $D(G(x)) = 0$, and accept the *real* data with $D(y) = 1$. For training, the loss for the discriminator is thus given by a sigmoid cross entropy for the two classes "generated" and "real":

$$\begin{aligned}\mathcal{L}_D(D,G) =& \mathbb{E}_{y\sim p_{\mathrm{y}}(y)}[-\log D(y)] + \mathbb{E}_{x\sim p_{\mathrm{x}}(x)}[-\log(1-D(G(x)))] \\ =& \mathbb{E}_m[-\log D(y_m)] + \mathbb{E}_n[-\log(1-D(G(x_n)))]\,,\end{aligned} \tag{1}$$

where n is the number of drawn inputs x, while m denotes the number of real data samples y. Here we use the notation $y \sim p_{\mathrm{y}}(y)$ for samples y being drawn from a corresponding probability data distribution p_{y}, which will later on be represented by our numerical simulation framework. The continuous distribution $\mathcal{L}_D(D,G)$ yields the average of discrete samples y_n and x_m in the second line of Eq. (1). We will omit the $y \sim p_{\mathrm{y}}(y)$ and $x \sim p_{\mathrm{x}}(x)$ subscripts of the sigmoid cross entropy, and n and m subscripts of $D(y_m)$ and $G(x_n)$, for clarity below.

In contrast to the discriminator, the generator is trained to "fool" the discriminator into accepting its samples and thus to generate output that is close to the real data from y. In practice, this means that the generator is trained to drive the discriminator result for its outputs to one. Instead of directly using the negative discriminator loss, GANs typically use

$$\mathcal{L}_G(D,G) = \mathbb{E}_{x\sim p_{\mathrm{x}}(x)}[-\log(D(G(x)))] = \mathbb{E}_n[-\log(D(G(x)))] \tag{2}$$

as the loss function for the generator, in order to reduce diminishing gradient problems [Goodfellow 2016]. As D is realized as a NN, it is guaranteed to be sufficiently differentiable as a loss function for G. In practice, both discriminator and generator are trained in turns and will optimally reach an equilibrium state.

As we target a super-resolution problem, our goal is not to generate an arbitrary high-resolution output, but one that corresponds to a low-resolution input, and hence we employ a *conditional* GAN. In terms of the dual optimization problem described above, this means that the input x now represents the low-resolution data set, and the discriminator is provided with x in order to establish and ensure the correct relationship between input and output, i.e., we now have $D(x,y)$ and $D(x,G(x))$ [Mirza and Osindero 2014]. Furthermore, previous work [Zhao et al. 2015] has shown that an additional L_1 loss term with a small weight can be added to the generator to ensure that its output stays close to the ground truth y. This yields $\lambda_{L_1}\mathbb{E}_n \left\|G(x)-y\right\|_1$, where λ_{L_1} controls the strength of this term, and we use $\mathbb{E}$ for consistency to denote the expected value, in this discrete case being equivalent to an average.

3.2 Loss in Feature Spaces

In order to further control the coupled, non-linear optimization process, the features of the underlying CNNs can be constrained. This is an important issue, as controlling the training process of GANs is known as a difficult problem. Here, we extend previous work on feature space losses, which were shown to improve realism in natural images [Dosovitskiy and Brox 2016], and were also shown to help with mode collapse problems [Salimans et al. 2016]. To achieve this goal, an L_2 loss over parts or the whole feature space of a neural network is introduced for the generator. I.e., the intermediate results of the generator network are constrained w.r.t. a set of intermediate reference data. While previous work typically makes use of manually selected layers of pre-trained networks, such as the VGG net, we propose to use features of the discriminator as constraints instead.

Thus, we incorporate a novel loss term of the form

$$\mathcal{L}_f = \mathbb{E}_{n,j}\lambda_f^j \left\|F^j(G(x)) - F^j(y)\right\|_2^2\,, \tag{3}$$

where j is a layer in our discriminator network, and F^j denotes the activations of the corresponding layer. The factor λ_f^j is a weighting term, which can be adjusted on a per layer basis, as we will discuss in Sec. 5.2. It is particularly important in this case that we can employ the discriminator here, as no suitable, pre-trained networks are available for three-dimensional flow problems.

Interestingly, these weights yields different and realistic results both for positive as well as negative choices for the weights. For $\lambda_f > 0$ these loss terms effectively encourage a minimization of the mean feature space distances of real and generated data sets, such that generated features resemble features of the reference. Surprisingly, we found that training runs with $\lambda_f < 0$ also yield excellent, and often slightly better results. As we are targeting conditional GANs, our networks are highly constrained by the inputs. Our explanation for this behavior is that a negative feature loss in this setting encourages the optimization to generate results that differ in terms of the features, but are still similar, ideally indistinguishable, in terms of their final output. This is possible as we are not targeting a single ground-truth result, but rather, we give the generator the freedom to generate any result that best fits the collection of inputs it receives. From our experience, this loss term drives the generator towards realistic detail, an example of which can be seen in Fig. 3. Note that due to the non-linear nature of the optimization, linearly changing λ_f yields to models with significant differences in the generated small scale features.

3.3 Temporal Coherence

While the GAN process described so far is highly successful at generating highly detailed and realistic outputs for static frames, these details are particularly challenging in terms of their temporal coherence. Since both the generator and the discriminator work on

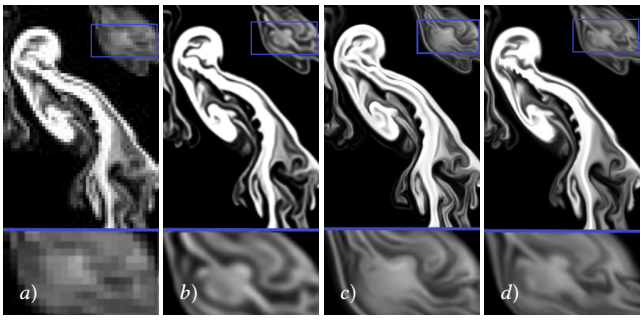

Fig. 3. From left to right: a) a sample, low-resolution input, b) a CNN output with naive L_2 loss (no GAN training), c) our tempoGAN output, and d) the high-resolution reference. The L_2 version learns a smooth result without small scale details, while our output in (c) surpasses the detail of the reference in certain regions.

every frame independently, subtle changes of the input x can lead to outputs $G(x)$ with distinctly different details for higher spatial frequencies.

When the ground truth data y comes from a transport process, such as frame motion or flow motion, it typically exhibits a very high degree of temporal coherence, and a velocity field v_y exists for which $y^t = \mathcal{A}(y^{t-1}, v_y^{t-1})$. Here, we denote the advection operator (also called warp or transport in other works) with $\mathcal{A}$, and we assume without loss of generality that the time step between frame t and $t-1$ is equal to one. Discrete time steps will be denoted by superscripts, i.e., for a function y of space and time $y^t = y(\mathbf{x}, t)$ denotes a full spatial sample at time t. Similarly, in order to solve the temporal coherence problem, the relationship $G(x^t) = \mathcal{A}(G(x^{t-1}), v^{t-1}_{G(x)})$ should hold, which assumes that we can compute a motion $v_{G(x)}$ based on the generator input x. While directly computing such a motion can be difficult and unnecessary for general GAN problems, we can make use of the ground truth data for y in our conditional setting. I.e., in the following, we will use a velocity reference v_y corresponding to the target y, and perform a spatial down-sampling to compute the velocity v_x for input x.

Equipped with v_x, one possibility to improve temporal coherence would be to add an L_2 loss term of the form:

$$\mathcal{L}_{2,t} = \|G(x^t) - \mathcal{A}(G(x^{t-1}), v_x^{t-1})\|_2^2 \tag{4}$$

We found that extending the forward-advection difference with backward-advection improves the results further, i.e., the following L_2 loss is clearly preferable over Eq. (4):

$$\mathcal{L}_{2,t} = \|G(x^t) - \mathcal{A}(G(x^{t-1}), v_x^{t-1})\|_2^2 + \|G(x^t) - \mathcal{A}(G(x^{t+1}), -v_x^{t+1})\|_2^2 \tag{5}$$

, where we align the next frame at $t+1$ by advecting with $-v_x^{t+1}$.

While this $\mathcal{L}_{2,t}$ based loss improves temporal coherence, our tests show that its effect is relatively small. E.g., it can improve outlines, but leads to clearly unsatisfactory results, which are best seen in the accompanying video. One side effect of this loss term is that it can easily be minimized by simply reducing the values of $G(x)$. This is visible, e.g., in the second column of Fig. 4, which contains noticeably less density than the other versions and the ground truth. However, we do not want to drive the generator towards darker outputs, but rather make it aware of how the data should change over time.

Instead of manually encoding the allowed temporal changes, we propose to use another discriminator D_t, that learns from the given data whose changes are admissible. In this way, the original spatial discriminator, which we will denote as $D_s(x, G(x))$ from now on, guarantees that our generator learns to generate realistic details, while the new temporal discriminator D_t mainly focuses on driving $G(x)$ towards solutions that match the temporal evolution of the ground-truth y.

Specifically, D_t takes three frames as input. We will denote such sets of three frames with a tilde in the following. As real data for the discriminator, the set $\widetilde{Y}_{\mathcal{A}}$ contains three consecutive and advected frames, thus $\widetilde{Y}_{\mathcal{A}} = \{\mathcal{A}(y^{t-1}, v_x^{t-1}), y^t, \mathcal{A}(y^{t+1}, -v_x^{t+1})\}$. The generated data set contains correspondingly advected samples from the generator: $\widetilde{G}_{\mathcal{A}}(\widetilde{X}) = \{\mathcal{A}(G(x^{t-1}), v_x^{t-1}), G(x^t), \mathcal{A}(G(x^{t+1}), -v_x^{t+1})\}$.

Similar to our spatial discriminator D_s, the temporal discriminator D_t is trained as a binary classifier on the two sets of data:

$$\mathcal{L}_{D_t}(D_t, G) = \mathbb{E}_m[-\log D_t\left(\widetilde{Y}_{\mathcal{A}}\right)] + \mathbb{E}_n[-\log(1 - D_t\left(\widetilde{G}_{\mathcal{A}}\left(\widetilde{X}\right)\right))] \tag{6}$$

, where set $\widetilde{X}$ also contains three consecutive frames, i.e., $\widetilde{X} = \{x^{t-1}, x^t, x^{t+1}\}$. Note that unlike the spatial discriminator, D_t is not a conditional discriminator. It does not "see" the conditional input x, and thus D_t is forced to make its judgment purely based on the given sequence.

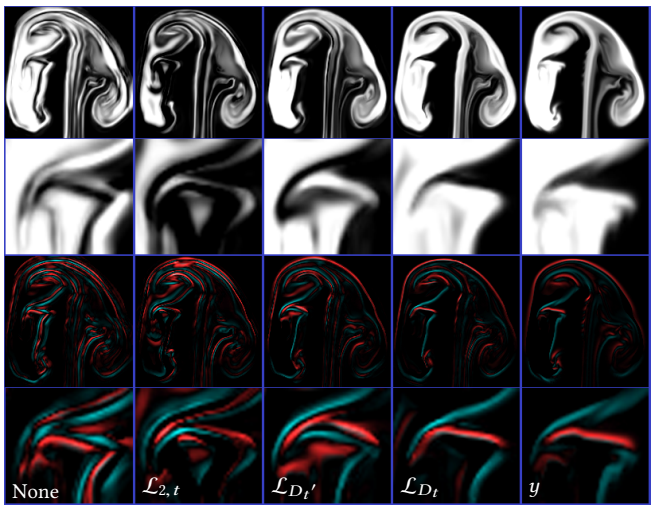

Fig. 4. A comparison of different approaches for temporal coherence. The top two rows show the inferred densities, while the bottom two rows contain the time derivative of the frame content computed with a finite difference between frame t and $t+1$. Positive and negative values are color-coded with red and blue, respectively. From left to right: no temporal loss applied, $\mathcal{L}_{2,t}$ loss applied, $\mathcal{L}_{D_t'}$, i.e., applied without advection, $\mathcal{L}_{D_t}$ applied with advection (our full tempoGAN approach), and the ground-truth y. From left to right across the different versions, the derivatives become less jagged and less noisy, as well as more structured and narrow. This means the temporal coherence is improved, esp. for the result from our algorithm ($\mathcal{L}_{D_t}$).

In Fig. 4, we show a comparison of the different loss variants for improving temporal coherence. The first column is generated with only the spatial discriminator, i.e., provides a baseline for the improvements. The second column shows the result using the L_2-based temporal loss $\mathcal{L}_{2,t}$ from Eq. (5), while the fourth column shows the result using D_t from Eq. (6). The last column is the ground-truth data y. The first two rows show the generated density fields. While $\mathcal{L}_{2,t}$ reduces overall density content, the result with D_t is clearly closer to the ground truth. The bottom two rows show time derivatives of the densities for frames t and t+1. Again, the result from D_t and the ground-truth y match closely in terms of their time derivatives. The large and jagged values of the first two rows indicate the undesirable temporal changes produced by the regular GAN and the $\mathcal{L}_{2,t}$ loss.

In the third column of Fig. 4, we show a simpler variant of our temporal discriminator. Here, we employ the discriminator without aligning the set of inputs with advection operations, i.e.,

$$\mathcal{L}_{D'_t}(D'_t, G) = \mathbb{E}_m[-\log D_t\left(\widetilde{Y}\right)] + \mathbb{E}_n[-\log(1 - D_t(\widetilde{G}\left(\widetilde{X}\right)))] \tag{7}$$

with $\widetilde{Y} = \{y^{t-1}, y^t, y^{t+1}\}$ and $\widetilde{G}\left(\widetilde{X}\right) = \{G(x^{t-1}), G(x^t), G(x^{t+1})\}$.

This version improves results compared to $\mathcal{L}_{2,t}$, but does not reach the level of quality of $\mathcal{L}_{D_t}$, as can be seen in Fig. 4. Additionally, we found that $\mathcal{L}_{D_t}$ often exhibits a faster convergence during the training process. This is an indication that the underlying neural networks have difficulties aligning and comparing the data by themselves when using $\mathcal{L}_{D'_t}$. This intuition is illustrated in Fig. 5, where we show example content of the regular data sets $\widetilde{Y}$ and the advected version $\widetilde{Y}_{\mathcal{A}}$ side by side. In this figure, the three chronological frames are visualized as red, green, and blue channels of the images. Thus, a pure gray-scale image would mean perfect alignment, while increasing visibility of individual colors indicates un-aligned features in the data. Fig. 5 shows that, although not perfect, the advected one leads to clear improvements in terms of aligning the features of the data sets, despite only using the approximated coarse velocity fields v_x. Our experiments show that this alignment successfully improves the backpropagated gradients such that the generator learns to produce more coherent outputs. However, when no flow fields are available, $\mathcal{L}_{D'_t}$ still represents a better choice than the simpler $\mathcal{L}_{2,t}$ version. We see this as another indicator of the power of adversarial training models. It seems to be preferable to let a neural network learn and judge the specifics of a data set, instead of manually specifying metrics, as we have demonstrated for data sets of fluid flow motions above.

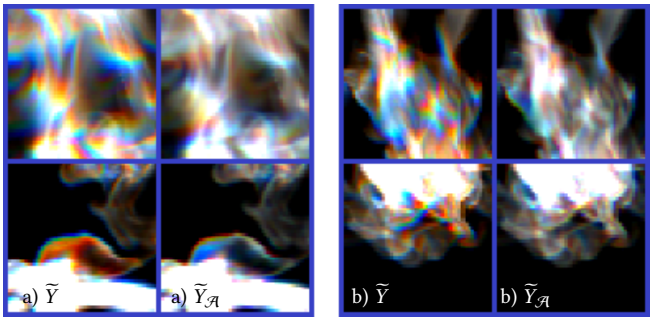

Fig. 5. These images highlight data alignment due to advection. Three consecutive frames are encoded as R, G, B channels of a single image, thus, ideally a fully aligned image would only contain shades of grey. The two rows contain front and top views in the top and bottom row, respectively. We show two examples, a) and b). Each of them contains $\widetilde{Y}$ left, and $\widetilde{Y}_{\mathcal{A}}$ right. The RGB channels are the three input frames, t-1, t, and t+1. Compared with $\widetilde{Y}$, $\widetilde{Y}_{\mathcal{A}}$ is significantly less saturated, i.e., better aligned.

It is worth pointing out that our formulation for D_t in Eq. (6) means that the advection step is an inherent part of the generator training process. While v_x can be pre-computed, it needs to be applied to the outputs of the generator during training. This in turn means that the advection needs to be tightly integrated into the training loop. The results discussed in the previous paragraph indicate that if this is done correctly, the loss gradients of the temporal discriminator are successfully passed through the advection steps to give the generator feedback such that it can improve its results. In the general case, advection is a non-linear function, the discrete approximation for which we have abbreviated with $\mathcal{A}(y^t, v_y^t)$ above. Given a known flow field v_y and time step, we can linearize this equation to yield a matrix $My = \mathcal{A}(y^t, v_y^t) = y^{t+1}$. E.g., for a first order approximation, M would encode the Euler-step lookup of source positions and linear interpolation to compute the solution. While we have found first order scheme (i.e., semi-Lagrangian advection) to work well, M could likewise encode higher-order methods for advection.

We have implemented this process as an advection layer in our network training, which computes the advection coefficients, and performs the matrix multiplication such that the discriminator receives the correct sets of inputs. When training the generator, the same code is used, and the underlying NN framework can easily compute the necessary derivatives. In this way, the generator actually receives three accumulated, and aligned gradients from the three input frames that were passed to D_t.

3.4 Full Algorithm

While the previous sections have explained the different parts of our final loss function, we summarize and discuss the combined loss in the following section. We will refer to our full algorithm as *tempoGAN*. The resulting optimization problem that is solved with NN training consists of three coupled non-linear sub-problems: the generator, the conditional spatial discriminator, and the un-conditional temporal discriminator. The generator has to effectively minimize both discriminator losses, additional feature space constraints, and a L_1 regularization term. Thus, the loss functions can be summarized as:

$$\begin{aligned}
\mathcal{L}_{D_t}(D_t, G) &= -\mathbb{E}_m[\log D_t(\widetilde{Y}_{\mathcal{A}})] - \mathbb{E}_n[\log\left(1 - D_t\left(\widetilde{G}_{\mathcal{A}}\left(\widetilde{X}\right)\right)\right)] \\
\mathcal{L}_{D_s}(D_s, G) &= -\mathbb{E}_m[\log D_s(x, y)] - \mathbb{E}_n[\log(1 - D_s(x, G(x)))] \\
\mathcal{L}_G(D_s, D_t, G) &= -\mathbb{E}_n[\log D_s(x, G(x))] - \mathbb{E}_n[\log D_t\left(\widetilde{G}_{\mathcal{A}}\left(\widetilde{X}\right)\right)] \\
&\quad + \mathbb{E}_{n,j}\lambda_f^j \left\|F^j(G(x)) - F^j(y)\right\|_2^2 + \lambda_{L_1}\mathbb{E}_n \left\|G(x) - y\right\|_1
\end{aligned} \tag{8}$$

Our generator has to effectively compete against two powerful adversaries, who, along the lines of "the enemy of my enemy is my friend", implicitly cooperate to expose the results of the generator.

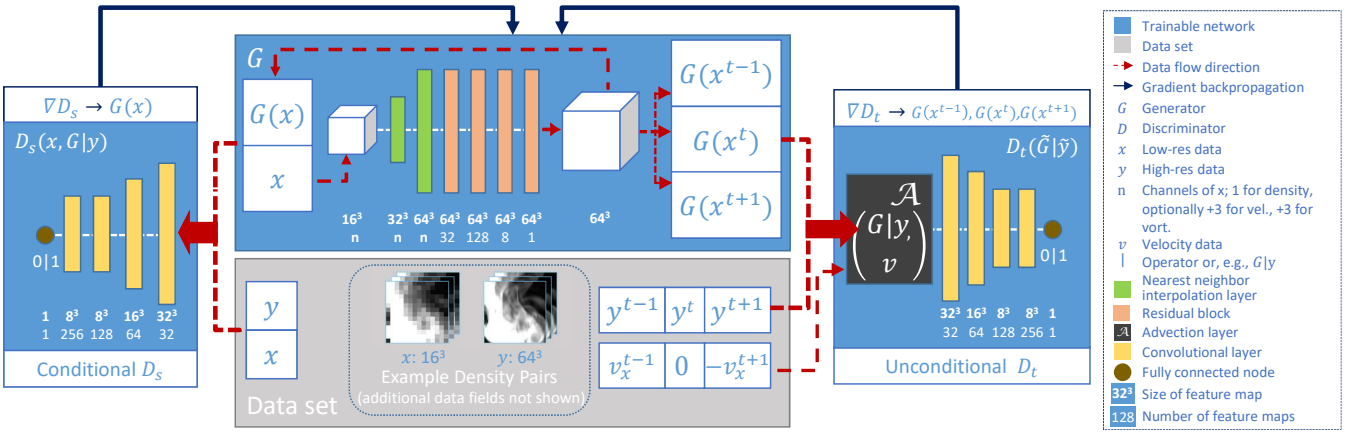

Fig. 6. Here an overview of our tempoGAN architecture is shown. The three neural networks (blue boxes) are trained in conjunction. The data flow between them is highlighted by the red and black arrows. Note that x and y denote fluid data that contains velocity and/or vorticity fields, as well as density depending on the chosen architecture (see Sec. 4.2).

E.g., we have performed tests without D_s, only using D_t, and the resulting generator outputs were smooth in time, but clearly less detailed than when using both discriminators.

Among the loss terms of the generator, the L_1 term has a relatively minor role to stabilize the training by keeping the averaged output close to the target. However, due to the complex optimization problem, it is nonetheless helpful for successful training runs. The feature space loss, on the other hand, directly influences the generated features. In the adversarial setting the discriminator most likely learns distinct features that only arise for the ground truth (positive features), or those that make it easy to identify generated versions, i.e., negative features that are only produced by the generator. Thus, while training, the generator will receive gradients to make it produce more features of the targets from $F(y)$, while the gradients from $F(G(x))$ will penalize the generation of recognizable negative features.

While positive values for λ_f reinforce this behavior, it is less clear why negative values can lead to even better results in certain cases. Our explanation for this behavior is that the negative weights drive the generator towards distinct features that have to adhere to the positive and negative features detected by the discriminator, as explained above in Sec. 3.2, but at the same time differ from the average features in y. Thus, the generator cannot simply create different or no features, as the discriminator would easily detect this. Instead it needs to develop features that are like the ones present in the outputs y, but don't correspond to the average features in $F(y)$, which, e.g., leads to the fine detailed outputs shown in Fig. 3.

4 ARCHITECTURE AND TRAINING DATA

While our loss function theoretically works with any realization of G, D_s and D_t, their specifics naturally have significant impact on performance and the quality of the generated outputs. A variety of network architectures has been proposed for training generative models [Berthelot et al. 2017; Goodfellow et al. 2014; Radford et al. 2016], and in the following, we will focus on pure convolutional networks for the generator, i.e., networks without any fully connected layers. A fully convolutional network has the advantage that the trained network can be applied to inputs of arbitrary sizes later on. We have experimented with a large variety of generator architectures, and while many simpler networks only yielded sub-optimal results, we have achieved high quality results with generators based on the popular U-net [Isola et al. 2017; Ronneberger et al. 2015], as well as with residual networks (*res-nets*) [Lim et al. 2017]. The U-net concatenates activations from earlier layers to later layers (so called *skip connections*) in order to allow the network to combine high- and low-level information, while the res-net processes the data using by multiple *residual blocks*. Each of these residual blocks convolves the inputs without changing their spatial size, and the result of two convolutional layers is added to the original signal as a "residual" correction. In the following, we will focus on the latter architecture, as it gave slightly sharper results in our tests.

We found the discriminator architecture to be less crucial. As long as enough non-linearity is introduced over the course of several hidden layers, and there are enough weights, changing the connectivity of the discriminator did not significantly influence the generated outputs. Thus, in the following, we will always use discriminators with

Alg. 1 tempoGAN training algorithm

for number of training steps **do**
  for k_{D_s} **do**
    Compute data-augmented mini batch x, y
    Update D_s with $\nabla_{D_s}[\mathcal{L}_{D_s}(D_s, G)]$
  for k_{D_t} **do**
    Compute data-augmented mini batch $\widetilde{X}$, $\widetilde{Y}$
    Compute advected frames $\widetilde{Y}_{\mathcal{A}}$ and $\widetilde{G}_{\mathcal{A}}\left(\widetilde{X}\right)$
    Update D_t with $\nabla_{D_t}[\mathcal{L}_{D_t}(D_t, G)]$
  for k_G **do**
    Compute data-augmented mini batch x, y, $\widetilde{X}$
    Compute advected frames $\widetilde{G}_{\mathcal{A}}\left(\widetilde{X}\right)$
    Update G with $\nabla_G[\mathcal{L}_G(D_s, D_t, G)]$

four convolutional layers with leaky ReLU activations[1] followed by a fully connected layer to output the final score. As suggested by Odena et al. [2016], we use the nearest-neighbor interpolation layers as the first two layers in our generator, instead of deconvolutional ones, and in the discriminator networks, the kernel size is divisible by the corresponding stride. An overview of the architecture of our neural networks is shown in Fig. 6, while their details, such as layer configuration and activation functions, can be found in Appendix A.

4.1 Data Generation and Training

We use a randomized smoke simulation setup to generate the desired number of training samples. For this we employ a standard fluids solve [Stam 1999] with MacCormack advection and MiC-preconditioned CG solver. We typically generate around 20 simulations, with 120 frames of output per simulation. For each of these, we randomly initialize a certain number of smoke inflow regions, another set of velocity inflows, and a randomized buoyancy force. As inputs x, we use a down-sampled version of the simulation data sets, typically by a factor of 4, while the full resolution data is used as ground truth y. Note that this setup is inherently *multi-modal*: for a single low resolution configuration, an infinitely large number of correct high resolution exists. We do not explicitly sample the high resolution solution space, but the down-sampling in conjunction with data augmentation lead to ambiguous low- and high-resolution pairs of input data. To prevent a large number of primarily empty samples, we discard inputs with average smoke density of less than 0.02. Details of the parametrization can be found in Appendix B, and visualizations of the training data sets can be found in the supplemental video. In addition, we show examples generated from a two-dimensional rising smoke simulation with a different simulation setup than the one used for generating the training data. It is, e.g., used in Fig. 3.

We use the same modalities for all training runs: we employ the commonly used ADAM optimizer[2] with an initial learning rate of $2 \cdot 10^{-4}$ that decays to 1/20th for second half of the training iterations. All parameters were determined experimentally, details are given in Appendix B. The number of training iterations is typically on the order of 10k. We use 20% of the data for testing and the remaining 80% for training. Our networks did not require any additional regularization such as dropout or weight decay. The training procedure is summarized again in Alg. 1. Due to the typically limited amount of GPU memory, especially for 3D data sets, we perform multiple training steps for each of the components. Detail are listed in Appendix B. In Alg. 1, we use k_{D_s}, k_{D_t}, and k_G to denote the number training iterations for D_s, D_t, and G, respectively.

While the coupled non-linear optimization can yield different results even for runs with the same parameters due to the non-deterministic nature of parallelized operations, we found the results to be stable in terms of quality. In particular, we did not find it necessary to change the weights of the different discriminator loss terms. However, if desired, λ_f can be used to influence the learned details as described above. For training and running the trained networks, we use Nvidia GeForce GTX 1080 Ti GPUs (each with 11GB Ram) and Intel Core i7-6850K CPUs, while we used the *tensorflow* and *mantaflow* software frameworks for deep learning and fluid simulation implementations, respectively.

[1] With a leaky tangent of 0.2 for the negative half space.
[2] Parameterized with $\beta = 0.5$.

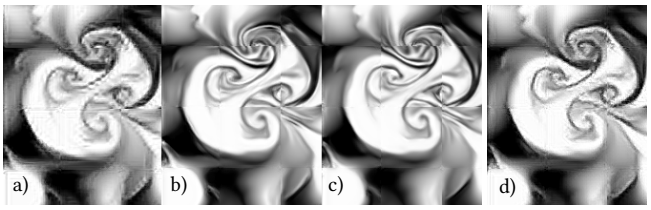

Fig. 7. An illustration of different training results after 40k iterations with different input fields: a) ρ, b) ρ + **v**, c) ρ + **v** + **w**, all with similar network sizes. Version d) with only ρ has 2x the number of weights. The seams in the images show the size of the training patches. Supplemental physical fields lead to clear improvements in b) and c), that even additional weights cannot compensate for.

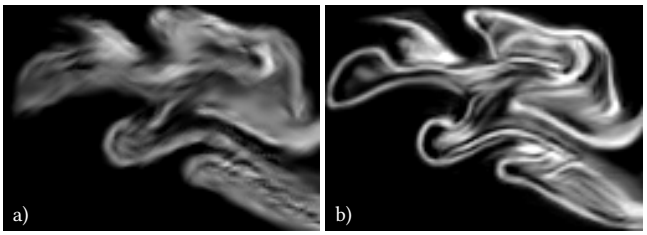

Fig. 8. An identical GAN network trained with the same set of input data While version a) did not use data augmentation, leading to blurry results with streak-like artifacts, version b), with data augmentation, produced sharp and detailed outputs.

4.2 Input Fields

On first sight, it might seem redundant and unnecessary to input flow velocity **v** and vorticity **w** in addition to the density ρ. After all, we are only interested in the final output density, and many works on GANs exist, which demonstrate that detailed images can be learned purely based on image content.

However, over the course of numerous training runs, we noticed that giving the networks additional information about the underlying physics significantly improves convergence and quality of the inferred results. An example is shown in Fig. 7. Here, we show how the training evolves for three networks with identical size, structure and parameters, the only difference being the input fields. From left to right, the networks receive (ρ), (ρ, **v**), and (ρ, **v**, **w**). Note that these fields are only given to the generator, while the discriminator always only receives (ρ) as input. The version with only density passed to the generator, $G(\rho)$, fails to reconstruct smooth and detailed outputs. Even after 40000 iterations, the results exhibit strong grid artifacts and lack detailed structures. In contrast, both versions with additional inputs start to yield higher quality outputs earlier during training. While adding **v** is crucial, the addition of **w** only yields subtle improvements (most apparent at the top of the images

in Fig. 7), which is why we will use $(\rho, \mathbf{v})$ to generate our final results below. The full training run comparison is in our supplemental video.

We believe that the insight that auxiliary fields help improving training and inference quality is a surprising and important one. The networks do not get any explicit guidance on how to use the additional information. However, it clearly not only learns to use this information, but also benefits from having this supporting information about the underlying physics processes. While larger networks can potentially alleviate the quality problems of the density-only version, as illustrated in Fig. 7 d), we believe it is highly preferable to instead construct and train smaller, physics-aware networks. This not only shortens training times and accelerates convergence, but also makes evaluating the trained model more efficient in the long run. The availability of physical inputs turned out to be a crucial addition in order to successfully realize high-dimensional GAN outputs for space-time data, which we will demonstrate in Sec. 5.

4.3 Augmenting Physical Data

Data augmentation turned out to be an important component of our pipeline due to the high dimensionality of our data sets and the large amount of memory they require. Without sufficient enough training data, the adversarial training yields undesirable results due to overfitting. While data augmentation is common practice for natural images [Dosovitskiy et al. 2016; Krizhevsky et al. 2012], we describe several aspects below that play a role for physical data sets.

The augmentation process allows us to train networks having millions of weights with data sets that only contain a few hundred samples without overfitting. At the same time, we can ensure that the trained networks respect the invariants of the underlying physical problems, which is crucial for the complex space-time data sets of flow fields that we are considering. E.g., we know from theory that solutions obey Galilean invariance, and we can make sure our networks are aware of this property not by providing large data sets, but instead by generating data with different inertial frames on the fly while training.

In order to minimize the necessary size of the training set without deteriorating the result quality, we generate modified data sets at training time. We focus on spatial transformations, which take the form of $\tilde{\mathbf{x}}(\mathbf{p}) = \mathbf{x}(A\mathbf{p})$, where $\mathbf{p}$ is a spatial position, and A denotes an 4×4 matrix. For applying augmentation, we distinguish three types of components of a data set:

- *passive*: these components can be transformed in a straight forward manner as described above. An example of passive components are the advected smoke fields ρ, shown in many of our examples.
- *directional*: the content of these components needs to be transformed in conjunction with the augmentation. A good example is velocity, whose directions need to be adjusted for rotations and flips, i.e., $\tilde{\mathbf{v}}(\mathbf{p}) = A_{3\times3}\mathbf{v}(A\mathbf{p})$, where $A_{3\times3}$ is the upper left 3×3 matrix of A.
- *derived*: finally, derived components would be invalid after applying augmentation, and thus need to be re-computed. A good example are physical quantities such as vorticity, which contain mixed derivatives that cannot be easily transformed into a new frame of reference. However, these quantities typically can be calculated anew from other fields after augmentation.

If the data set contains quantities that cannot be computed from other augmented fields, this unfortunately means that augmentation cannot be applied easily. However, we believe that a large class of typical physics data sets can in practice be augmented as described here.

For matrix A, we consider affine transformation matrices that contain combinations of randomized translations, uniform scaling, reflections, and rotations. Here, only those transformations are allowed that do not violate the physical model for the data set. While shearing and non-uniform scaling could easily be added, they violate the NS momentum equation and thus should not be used for flow data. We have used values in the range $[0.85, 1.15]$ for scaling, and rotations by $[-90, 90]$ degrees. We typically do not load derived components into memory for training, as they are re-computed after augmentation. Thus, they are computed on the fly for a training batch and discarded afterwards.

The outputs of our simulations typically have significantly larger size than the input tiles that our networks receive. In this way, we have many choices for choosing offsets, in order to train the networks for shift invariance. This also aligns with our goal to train a network that will later on work for arbitrarily sized inputs. We found it important to take special care at spatial boundaries of the tiles. While data could be extended by Dirichlet or periodic boundary conditions, it is important that the data set boundaries after augmentation do not lie outside the original data set. We enforce this by choosing suitable translations after applying the other transformations. This ensures that all data sets contain only valid content, and the network does not learn from potentially unphysical or unrepresentative data near boundaries. We also do not augment the time axis in the same way as the spatial axes. We found that the spatial transformations above applied to velocity fields give enough variance in terms of temporal changes. An example of the huge difference that data augmentation can make is shown in Fig. 8. Here we compare two runs with the same amount of training data (160 frames of data), one with, the other one without data augmentation. While training a GAN directly with this data produces blurry results, the network converges to a final state with significantly sharper results with data augmentation. The possibility to successfully train networks with only a small amount of training data is what makes it possible to train networks for 3D+time data, as we will demonstrate in Sec. 5.

5 RESULTS AND APPLICATIONS

In the following, we will apply our method discussed so far to different data sets, and explore different application settings. Among others, we will discuss related topics such as art direction, training convergence, and performance. [3]

5.1 3D Results

We have primarily used the 2D rising plume example in the previous sections to ensure the different variants can be compared easily. In

[3] We will make code and trained models available upon acceptance of our work.

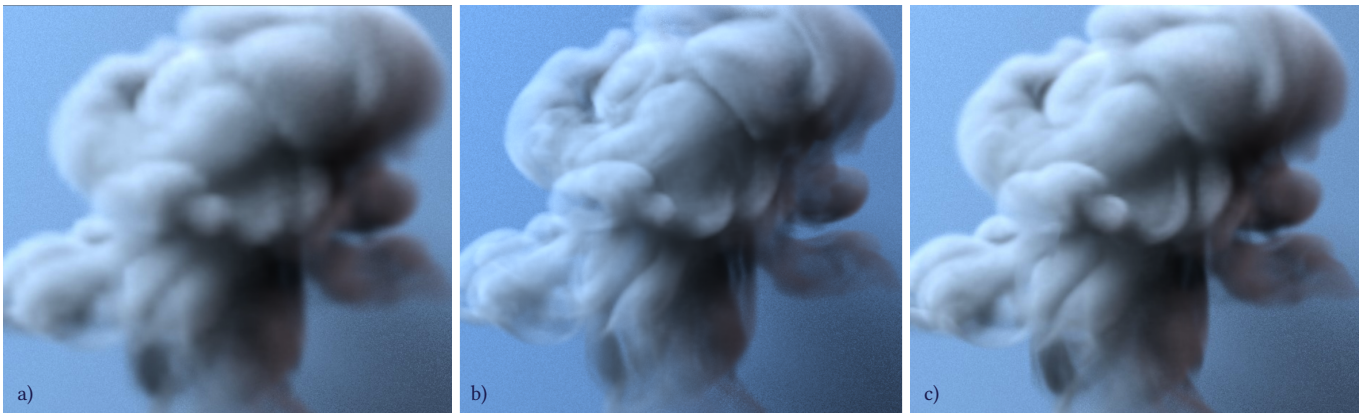

Fig. 9. These images show our algorithm applied to a 3D volume. F.l.t.r.: a). a coarse input volume (down-sampled from the reference c, rendered with cubic up-sampling), b). our result, and c). the high resolution reference. As in 2D, our trained model generates sharp features and detailed sheets that are at least on par with the reference.

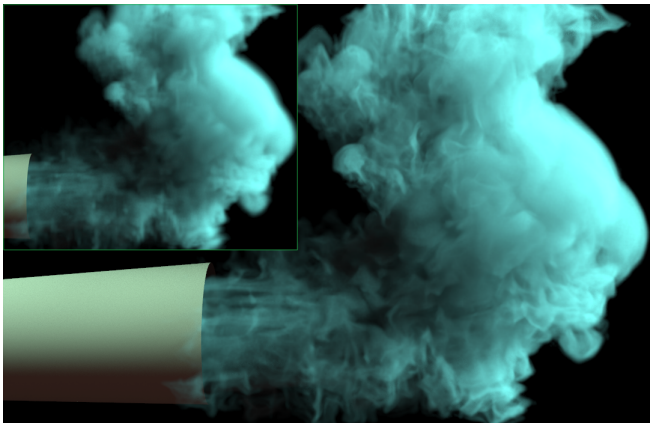

Fig. 10. We apply our algorithm to a horizontal jet of smoke in this example. The inset shows the coarse input (rendered with cubic up-sampling), and the result of our algorithm. The diffuse streaks caused by procedural turbulence in the input (esp. near the inflow) are turned into detailed wisps of smoke by our algorithm.

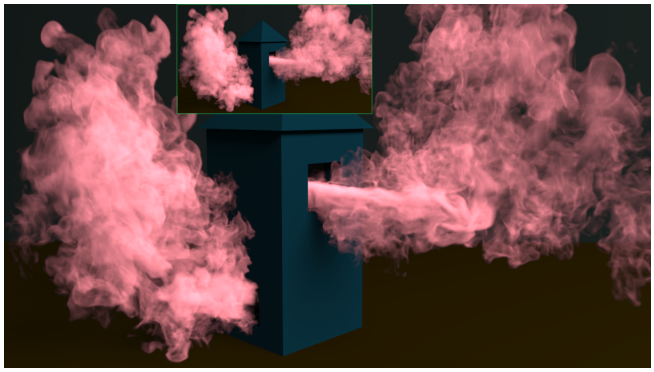

Fig. 11. Our algorithm generated a high-resolution volume around an obstacle with a final resolution of $1024 \times 720 \times 720$. The inset shows the input volume. This scene is also shown in Fig. 1 with a different visualization.

Fig. 9, we demonstrate that these results directly extend to 3D. We apply our method to a three-dimensional plume with resolution 64^3, which in this case was generated by down-sampling a 256^3 simulation such that we can compare our result to this reference solution. For this input data, the 256^3 output produced by our tempoGAN exhibits small scale features that are at least as detailed as the ground truth reference. The temporal coherence is especially important in this setting, which is best seen in the accompanying video.

We also apply our trained 3D model to two different inputs with higher resolutions. In both cases, we use a regular simulation augmented with additional turbulence to generate an interesting set of inputs for our method. A first scene with $150 \times 100 \times 100$ is shown in Fig. 10, where we generate a $600 \times 400 \times 400$ output with our method. The output closely resembles the input volumes, but exhibits a large number of fine details. Note that our networks were only trained with down-sampled inputs, but our models generalize well to regular simulation inputs without re-sampling, as illustrated by this example.

Our method also has no problems with obstacles in the flow, as shown in Fig. 11. This example has resolutions of $256 \times 180 \times 180$ and $1024 \times 720 \times 720$ for input and output volumes. The small-scale features closely adhere to the input flow around the obstacle. Although the obstacle is completely filled with densities towards the end of the simulation, there are no leaking artifacts as our method is applied independently to each input volume in the sequence. When showing the low-resolution input, we always employ cubic up-sampling, in order to not make the input look unnecessarily bad.

5.2 Fine Tuning Results

GANs have a reputation for being particularly hard to influence and control, and influencing the outcome of simulation results is an important topic for applications in computer graphics. In contrast

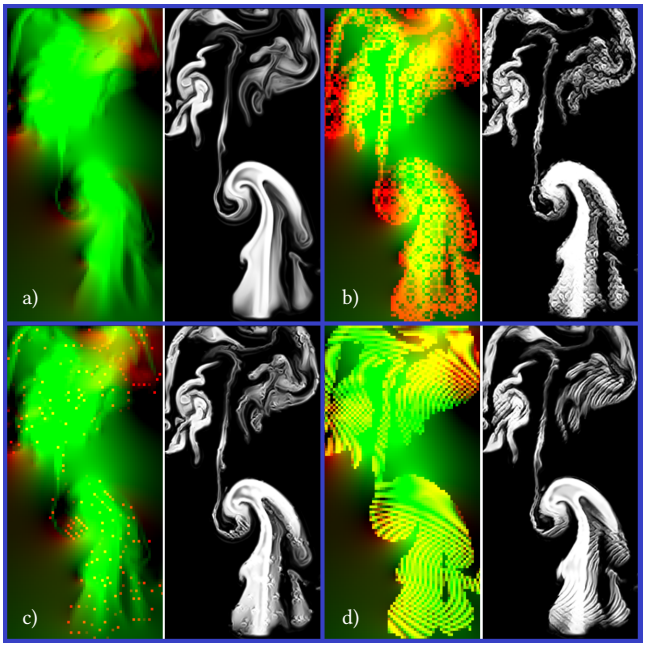

Fig. 12. The red&green images on the left of each pair represent the modified velocity inputs, while the corresponding result is shown on the right. For reference, pair a) shows the unmodified input velocity, and the regular output of our algorithm.

to procedural methods, regular GAN models typically lack intuitive control knobs to influence the generated results. While we primarily rely on traditional guiding techniques to control the low-resolution input, our method offers different ways to adjust the details produced by our tempoGAN algorithm.

A first control knob for fine-tuning the results is to modify the data fields of the conditional inputs. As described in Sec. 4.2, our generator receives the velocity in addition to the density, and it internally builds tight relationships between the two. We can use these entangled inputs to control the features produced in the outputs. To achieve this, we modify the velocity components passed to the generator with various procedural functions. Fig. 12 shows the results of original input and several modified velocity examples and the resulting density configurations. We have also experimented with noise fields instead [Mirza and Osindero 2014], but found that the trained networks completely ignored these fields. Instead, the strongly correlated velocity fields naturally provide a much more meaningful input for our networks, and as a consequence provide means for influencing the results.

In addition, Fig. 13 demonstrates that we can effectively suppress the generation of small scale details by setting all velocities to zero. Thus, the network learns a correlation between velocity magnitudes and amount of features. This is another indicator that the network learns to extract meaningful relationships from the data, as we expect turbulence and small-scale details to primarily form in regions with large velocities. Three-dimensional data can similarly be controlled, as illustrated in Fig. 14.

Fig. 13. An illustration how the entangled inputs of density and velocity can be used to fine tune the results: on the left the velocities were scaled up by a factor of 2, while the right hand side was scaled by zero. The network has learned a relationship between detail and velocities, leading to reduced details in regions where the velocity was set to zero.

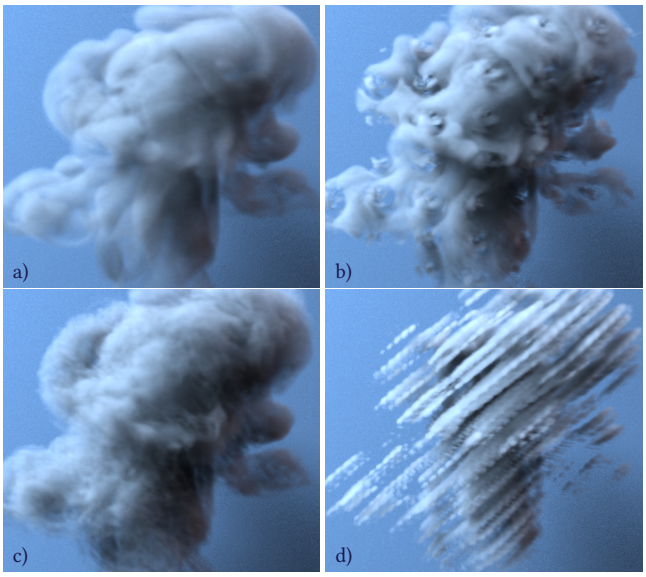

Fig. 14. a) is the result of tempoGAN with velocity set to zero. The other three examples were generated with modified velocity inputs to achieve more stylized outputs.

In Sec. 3.2, we discussed the influence of the λ_f parameter for small scale features. For situations where we might not have additional channels such as the velocity above, we can use λ_f to globally let the network generate different features. However, as this only provides a uniform change that is encoded in the trained network, the resulting differences are more subtle than those from the velocity modifications above. Examples of different 2D and 3D outputs can be found in Fig. 15 and Fig. 16, respectively.

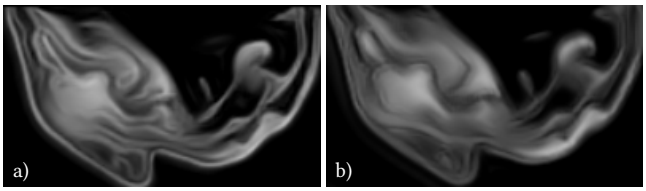

Fig. 15. A comparison of training runs with different feature loss weights: a) $\lambda_f^{1,\dots,4} = -10^{-5}$, b) $\lambda_f^{1,4} = 1/3 \cdot 10^{-4}$, $\lambda_f^{2,3} = -1/3 \cdot 10^{-4}$.

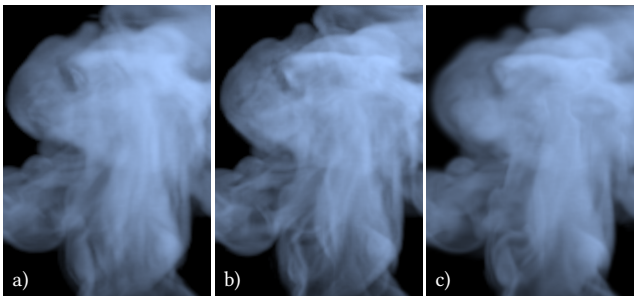

Fig. 16. A comparison of training runs with different feature loss weights in 3D: a) with $\lambda_f^{1,\dots,4} = -1/3 \cdot 10^{-6}$, b) with $\lambda_f^{1} = 1/3 \cdot 10^{-6}$, $\lambda_f^{2,3,4} = -1/3 \cdot 10^{-6}$. The latter yields a sharpened result. Image c) shows the high resolution reference.

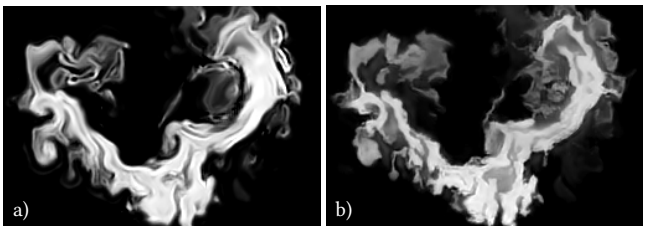

Fig. 17. Our regular model a) and one trained with wavelet turbulence data b). In contrast to the model trained with real simulation data, the wavelet turbulence model produces flat regions with sharper swirls, mimicking the input data.

5.3 Additional Variants

In order to verify that our network can not only work with two- or three-dimensional data from a Navier-Stokes solver, we generated a more synthetic data set by applying strong wavelet turbulence to a 4× up-sampled input flow. We then trained our network with down-sampled inputs, i.e., giving it the task to learn the output of the wavelet turbulence algorithm. Note that a key difference here is that wavelet turbulence normally requires a full high-resolution advection over time, while our method infers high-resolution data sets purely based on low-resolution data from a single frame.

Our network successfully learns to generate structures similar to the wavelet turbulence outputs, shown in Fig. 17. However, this data set turned out to be more difficult to learn than the original fluid simulation inputs. The training runs required two times more training data than the regular simulation runs, and we used a feature loss of $\lambda_f^{1,\dots,4} = 10^{-5}$. We assume that these more difficult training conditions are caused by the more chaotic nature of the procedural turbulence, and the less reasonable correlations between density and velocity inputs. Note that despite using more wavelet turbulence input data, it is still a comparatively small data set.

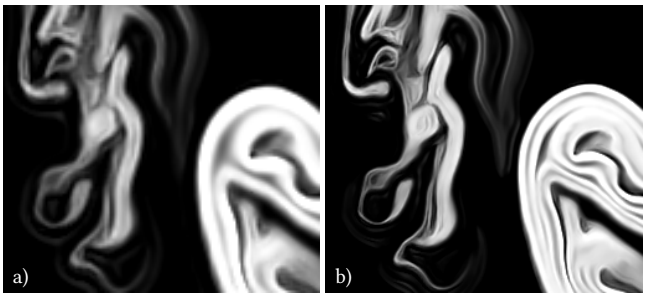

Fig. 18. a) is the network output after a single application. b) is the network recursively applied to a) with a scaling factor of 2, resulting in a total increase of 8×.

We additionally were curious how well our network works when it is applied to a generated output, i.e., a recursive application. The result can be found in Fig. 18, where we applied our network to its own output for an additional 2× upsampling. Thus, in total this led to an 8× increase of resolution. While the output is plausible, and clearly contains even more fine features such as thin sheets, there is a tendency to amplify features generated during the first application.

5.4 Training Progress

With the training settings given in Appendix B, our training runs typically converged to stable solutions of around 1/2 for the discriminator outputs after sigmoid activation. While this by itself does not guarantee that a desirable solution was found, it at least indicates convergence towards one of the available local minima.

However, it is interesting how the discriminator loss changes in the presence of the temporal discriminator. Fig. 19 shows several graphs of discriminator losses over the course of a full training run. Note that we show the final loss outputs from Eq. (1) and Eq. (6) here. A large value means the discriminator does "worse", i.e., it has more difficulty distinguishing real samples from the generated ones. Correspondingly, lower values mean it can separate them more successfully. In Fig. 19a) it is visible that the spatial discriminator loss decreases when the temporal discriminator is introduced. Here the graph only shows the spatial discriminator loss, and the discriminator itself is unchanged when the second discriminator is introduced. The training run corresponding to the green line is trained with only a spatial discriminator, and for the orange line with both spatial and temporal discriminators. Our interpretation of the lower loss for the spatial discriminator network is that the existence of a temporal discriminator in the optimization prevents the generator from using the part of the solution space with detailed, but flickering outputs. Hence, the generator is driven to find a solution from the temporally coherent ones, and as a consequence has a harder time, which in turn makes the job easier for the spatial

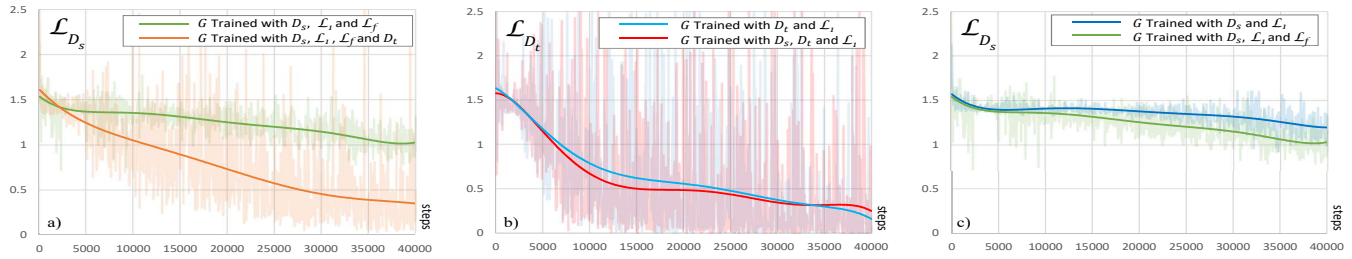

Fig. 19. Several discriminator loss functions over the course of the 40k training iterations. a) D_s (spatial discriminator) loss is shown in green without D_t, and orange with D_t. b) Temporal discriminator loss in blue with only D_t, and in red for tempoGAN (i.e., with D_s, and feature loss). c) Spatial discriminator loss is shown in green with $\mathcal{L}_f$, and in dark blue without. For each graph, the dark lines show smoothed curves. The full data is shown in a lighter color in the background.

discriminator. This manifests itself as a lower loss for the spatial discriminator, i.e. the lower orange curve in Fig. 19a).

Conversely, the existence of a spatial discriminator does not noticeably influence the temporal discriminator, as shown in Fig. 19b). This is also intuitive, as the spatial discriminator does not influence temporal changes. We found that a generator trained only with D_t typically produces fewer details than a generator trained with both. In conjunction, our tests indicate that the two discriminators successfully influence different aspects of the solution space, as intended. Lastly, Fig. 19c) shows that activating the negative feature loss from Sec. 3.2 makes the task for the generator slightly harder, resulting in a lowered spatial discriminator loss.

5.5 Performance

Training our two- and three-dimensional models is relatively expensive. Our full 2D runs typically take around 14 hours to complete (1 GPU), while the 3D runs took ca. 9 days using two GPUs. However, in practice, the state of the model after a quarter of this time is already indicative of the final performance. The remainder of the time is typically spent fine-tuning the network.

When using our trained network to generate high-resolution outputs in 3D, the limited memory of current GPUs poses a constraint on the volumes that can be processed at once, as the intermediate layers with their feature maps can take up significant amounts of memory. However, this does not pose a problem for generating larger final volumes, as we can subdivide the input volumes, and process them piece by piece. We generate tiles with a size of 136^3 on one GPU, with a corresponding input of size 34^3. Our 8 convolutional layers with a receptive field of 16 cells mean that up to four cells of an input could be influenced by a boundary. In practice, we found 3 input cells to be enough in terms of overlap. Generating a single 136^3 output took ca. 2.2 seconds on average. Thus, generating a 256^3 volume from an 64^3 input took 17.9s on average. Comparing the performance of our model with high resolution simulations is inherently difficult, due to the substantially different implementations and hardware platforms (CPU vs. GPU). However, for the example of Fig. 10 we estimate that fluid simulation at the full resolution would take ca. 31.5 minutes per frame of animation on average, while the evaluation of all volume tiles with our evaluation pipeline took ca. 3.9 minutes.

The cost for the trained model scales linearly with the number of cells in the volume, and in contrast to all previous methods for increasing the resolution of flow simulations, our method does not require any additional tracking information. It is also fully independent for all frames. Thus, our method could ideally be applied on the fly before rendering a volume, after which the high resolution data could be discarded. Additionally, due to GPU memory restrictions we currently evaluate our model in volumetric tiles with 3 cells of overlap for the input. This overlap can potentially be reduced further, and become unnecessary when enough memory is available to process the full input volume at once.

5.6 Limitations and Discussion

One limitation of our approach is that the network encodes a fixed resolution difference for the generated details. While the initial up-sampling layers can be stripped, and the network could thus be applied to inputs of any size, it will be interesting to explore different up-sampling factors beyond the factor of four which we have used throughout. With our current implementation, our method can also be slower than, e.g., calculating the advection for a high resolution grid. However, a high-res advection would typically not lead to different dynamics than those contained in the input flow, and require a sequential solve for the whole animation sequence. Our networks have so far also focused on buoyant smoke clouds. While obstacle interactions worked in our tests, we assume that networks trained for larger data sets and with other types of interactions could yield even better results.

Our three-dimensional networks needed a long time to train, circa nine days for our final model. Luckily, this is a one-time cost, and the network can be flexibly reused afterwards. However, if the synthesized small-scale features need to be fine-tuned, which we luckily did not find necessary for our work, the long runtimes could make this a difficult process. The feature loss weights clearly also are data dependent, e.g., we used different settings for simulation and wavelet turbulence data. Here, it will be an interesting direction for future work to give the network additional inputs for fine tuning the results beyond the velocity modifications which discussed in Sec. 5.2.

6 CONCLUSIONS

We have realized a first conditional GAN approach for four-dimensional data sets, and we have demonstrated that it is possible to train generators that preserve temporal coherence using our novel time discriminator. The network architecture of this temporal discriminator, which ensures that the generator receives gradient information even for complex transport processes, makes it possible to robustly train networks for temporal evolutions. We have shown that this discriminator improves the generation of stable details as well as the learning process itself. At the same time, our fully convolutional networks can be applied to inputs of arbitrary size, and our approach provides basic means for art direction of the generated outputs. We also found it very promising to see that our CNNs are able to benefit from coherent, physical information even in complex 3D settings, which led to reduced network sizes.

Overall, we believe that our contributions yield a robust and very general method for generative models of physics problems, and for super-resolution flows in particular. It will be highly interesting as future work to apply our tempoGAN to other physical problem settings, or even to non-physical data such as video streams.

ACKNOWLEDGMENTS

This work was funded by the ERC Starting Grant *realFlow* (StG-2015-637014). We would like to thank Wei He for helping with making the videos, and all members of the graphics labs of TUM, IST Austria and ETH Zurich for the thorough discussions.

REFERENCES

Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein GAN. *arXiv:1701.07875* (2017).

Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan NováK, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle. 2017. Kernel-predicting convolutional networks for denoising Monte Carlo renderings. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 97.

David Berthelot, Tom Schumm, and Luke Metz. 2017. BeGAN: Boundary equilibrium generative adversarial networks. *arXiv:1703.10717* (2017).

Prateep Bhattacharjee and Sukhendu Das. 2017. Temporal Coherency based Criteria for Predicting Video Frames using Deep Multi-stage Generative Adversarial Networks. In *Advances in Neural Information Processing Systems*. 4271–4280.

Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Chakravarty Alla Chaitanya, Anton Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 98.

Dongdong Chen, Jing Liao, Lu Yuan, Nenghai Yu, and Gang Hua. 2017. Coherent Online Video Style Transfer. In *The IEEE International Conference on Computer Vision (ICCV)*.

Mengyu Chu and Nils Thuerey. 2017. Data-Driven Synthesis of Smoke Flows with CNN-based Feature Descriptors. *ACM Trans. Graph.* 36(4), 69 (2017).

Emmanuel de Bezenac, Arthur Pajot, and Patrick Gallinari. 2017. Deep Learning for Physical Processes: Incorporating Prior Scientific Knowledge. *arXiv preprint arXiv:1711.07970* (2017).

Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. 2016. Image super-resolution using deep convolutional networks. *IEEE transactions on pattern analysis and machine intelligence* 38, 2 (2016), 295–307.

Alexey Dosovitskiy and Thomas Brox. 2016. Generating images with perceptual similarity metrics based on deep networks. In *Advances in Neural Information Processing Systems*. 658–666.

Alexey Dosovitskiy, Philipp Fischer, Jost Tobias Springenberg, Martin Riedmiller, and Thomas Brox. 2016. Discriminative unsupervised feature learning with exemplar convolutional neural networks. *IEEE Trans. Pattern Analysis and Mach. Int.* 38, 9 (2016), 1734–1747.

Amir Barati Farimani, Joseph Gomes, and Vijay S Pande. 2017. Deep Learning the Physics of Transport Phenomena. *arXiv:1709.02432* (2017).

John Flynn, Ivan Neulander, James Philbin, and Noah Snavely. 2016. DeepStereo: Learning to predict new views from the world's imagery. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 5515–5524.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proc. of IEEE Comp. Vision and Pattern Rec.* IEEE, 580–587.

Ian Goodfellow. 2016. NIPS 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160* (2016).

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. *stat* 1050 (2014), 10.

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2017. Image-to-image translation with conditional adversarial networks. *Proc. of IEEE Comp. Vision and Pattern Rec.* (2017).

Justin Johnson, Alexandre Alahi, and Li Fei-Fei. 2016. Perceptual losses for real-time style transfer and super-resolution. In *European Conference on Computer Vision*. Springer, 694–711.

Simon Kallweit, Thomas Müller, Brian McWilliams, Markus Gross, and Jan Novák. 2017. Deep Scattering: Rendering Atmospheric Clouds with Radiance-Predicting Neural Networks. *arXiv:1709.05418* (2017).

Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2017. Progressive growing of gans for improved quality, stability, and variation. *arXiv:1710.10196* (2017).

Ladislav Kavan, Dan Gerszewski, Adam W Bargteil, and Peter-Pike Sloan. 2011. Physics-inspired upsampling for cloth simulation in games. In *ACM Transactions on Graphics (TOG)*, Vol. 30. ACM, 93.

Byungmoon Kim, Yingjie Liu, Ignacio Llamas, and Jarek Rossignac. 2005. FlowFixer: Using BFECC for Fluid Simulation. In *Proceedings of the First Eurographics conference on Natural Phenomena*. 51–56.

Jiwon Kim, Jung Kwon Lee, and Kyoung Mu Lee. 2016. Accurate image super-resolution using very deep convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1646–1654.

Theodore Kim, Nils Thuerey, Doug James, and Markus Gross. 2008. Wavelet Turbulence for Fluid Simulation. *ACM Trans. Graph.* 27 (3) (2008), 50:1–6.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. NIPS, 1097–1105.

Lubor Ladicky, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. 2015. Data-driven fluid simulations using regression forests. *ACM Trans. Graph.* 34, 6 (2015), 199.

Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. 2016. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv:1609.04802* (2016).

Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. 2017. Enhanced deep residual networks for single image super-resolution. In *Proc. of IEEE Comp. Vision and Pattern Rec.*, Vol. 1. 3.

Ding Liu, Zhaowen Wang, Yuchen Fan, Xianming Liu, Zhangyang Wang, Shiyu Chang, and Thomas Huang. 2017. Robust Video Super-Resolution With Learned Temporal Dynamics. In *The IEEE International Conference on Computer Vision (ICCV)*.

Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. 2017. PDE-Net: Learning PDEs from Data. *arXiv:1710.09668* (2017).

Fujun Luan, Sylvain Paris, Eli Shechtman, and Kavita Bala. 2017. Deep Photo Style Transfer. *arXiv preprint arXiv:1703.07511* (2017).

W Magnus, F Henrik, A Chris, and M Stephen. 2011. Capturing Thin Features in Smoke Simulations. *Siggraph Talk* (2011).

Michael Mathieu, Camille Couprie, and Yann LeCun. 2015. Deep multi-scale video prediction beyond mean square error. *arXiv preprint arXiv:1511.05440* (2015).

Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. 2004. Fluid Control Using the Adjoint Method. *ACM Trans. Graph.* 23, 3 (2004), 449–456.

Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).

Lukas Mosser, Olivier Dubrule, and Martin J Blunt. 2017. Reconstruction of three-dimensional porous media using generative adversarial neural networks. *arXiv:1704.03225* (2017).

Rahul Narain, Jason Sewall, Mark Carlson, and Ming C. Lin. 2008. Fast Animation of Turbulence Using Energy Transport and Procedural Synthesis. *ACM Trans. Graph.* 27, 5 (2008), article 166.

Augustus Odena, Vincent Dumoulin, and Chris Olah. 2016. Deconvolution and Checkerboard Artifacts. *Distill* (2016). https://doi.org/10.23915/distill.00003

Zherong Pan, Jin Huang, Yiying Tong, Changxi Zheng, and Hujun Bao. 2013. Interactive Localized Liquid Motion Editing. *ACM Trans. Graph.* 32, 6 (Nov. 2013).

Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. 2017. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Trans. Graph.* 36, 4 (2017), 41.

Lukas Prantl, Boris Bonev, and Nils Thuerey. 2017. Pre-computed Liquid Spaces with Generative Neural Networks and Optical Flow. *arXiv:1704.07854* (2017).
Alec Radford, Luke Metz, and Soumith Chintala. 2016. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *Proc. ICLR* (2016).
Nick Rasmussen, Duc Quang Nguyen, Willi Geiger, and Ronald Fedkiw. 2003. Smoke simulation for large scale phenomena. In *ACM Transactions on Graphics (TOG)*, Vol. 22. ACM, 703–707.
Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 234–241.
Manuel Ruder, Alexey Dosovitskiy, and Thomas Brox. 2016. Artistic Style Transfer for Videos. In *Pattern Recognition - 38th German Conference, GCPR 2016, Hannover, Germany, September 12-15, 2016, Proceedings*. 26–36. https://doi.org/10.1007/978-3-319-45886-1_3
Masaki Saito, Eiichi Matsumoto, and Shunta Saito. 2017. Temporal generative adversarial nets with singular value clipping. In *IEEE International Conference on Computer Vision (ICCV)*. 2830–2839.
Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. 2016. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*. 2234–2242.
Hagit Schechter and Robert Bridson. 2008. Evolving sub-grid turbulence for smoke animation. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics symposium on Computer animation*. Eurographics Association, 1–7.
Andrew Selle, Ronald Fedkiw, Byungmoon Kim, Yingjie Liu, and Jarek Rossignac. 2008. An Unconditionally Stable MacCormack Method. *J. Sci. Comput.* 35, 2-3 (June 2008), 350–371.
Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556* (2014).
Jos Stam. 1999. Stable Fluids. In *Proc. ACM SIGGRAPH*. ACM, 121–128.
Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. 2016. Accelerating Eulerian Fluid Simulation With Convolutional Networks. *arXiv: 1607.03597* (2016).
Kiwon Um, Xiangyu Hu, and Nils Thuerey. 2017. Splash Modeling with Neural Networks. *arXiv:1704.04456* (2017).
Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient.. In *AAAI*. 2852–2858.
Hang Zhao, Orazio Gallo, Iuri Frosio, and Jan Kautz. 2015. Loss Functions for Neural Networks for Image Processing. *arXiv preprint arXiv:1511.08861* (2015).
Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. *arXiv:1703.10593* (2017).

A DETAILS OF NN ARCHITECTURES

To clearly specify our networks, we use the following notation. Let in(resolution, channels), out(resolution, output) present input and output information; NI(output-resolution) represent nearest-neighbor interpolation; C(output-resolution, filter size, output-channels) denote a convolutional layer. Our resolutions and filter sizes are the same for every spacial dimension for both 2D and 3D. Resolutions of feature maps are reduced when strides >1. We use RB to represent our residual blocks, and use C_S for adding residuals in a RB. E.g., RB_3 : [C_A(64, 5, 128), ReLU, C_B(64, 5, 128), BN] + [C_S(64, 1, 128), BN], ReLU means [($input \rightarrow C_A \rightarrow$ ReLU $\rightarrow C_B$) + ($input \rightarrow C_S$)] $\rightarrow$ ReLU, where + denotes element-wise addition. BN denotes batch normalization, which is not used in the last layer of G, the first layer of D_t and the first layer of D_s [Radford et al. 2016]. In addition, | denotes concatenation of layer outputs along the channel dimension.

Architectures of G, D_s and D_t:

G:
in(16, 4)
NI(64, 4)
RB_0 : [C_A(64, 5, 8), BN, ReLU, C_B(64, 5, 32), BN] + [C_S(64, 1, 32), BN], ReLU
RB_1 : [C_A(64, 5, 128), BN, ReLU, C_B(64, 5, 128), BN] + [C_S(64, 1, 128), BN], ReLU
RB_2 : [C_A(64, 5, 32), BN, ReLU, C_B(64, 5, 8), BN] + [C_S(64, 1, 8), BN], ReLU
RB_3 : [C_A(64, 5, 2), ReLU, C_B(64, 5, 1)] + [C_S(64, 1, 1)], ReLU
out(64, 1)

D_s:	D_t:
in_x(16, 1) , the conditional density NI(64, 1)|in_y(64, 1) , the high-res input to classify	in_y(64, 3), the 3 high-res frames to classify
C(32, 4, 32), leaky ReLU	C(32, 4, 32), leaky ReLU
C(16, 4, 64), BN, leaky ReLU	C(16, 4, 64), BN, leaky ReLU
C(8, 4, 128), BN, leaky ReLU	C(8, 4, 128), BN, leaky ReLU
C(8, 4, 256), BN, leaky ReLU	C(8, 4, 256), BN, leaky ReLU
Fully connected, σ activation	Fully connected, σ activation
out(1, 1)	out(1, 1)

B PARAMETERS & DATA STATISTICS

Below we summarize all parameters for training runs and data generation. Note that the model size includes compression, and we train the individual networks multiple times per iteration, as indicated below.

Details of generated results:

test	input size	tile (34^3) number	output size	time
Fig. 15 a)	128^2	-	512^2	0.064s/frame
-	34^3	1	136^3	2.2s/frame
Fig. 9 b)	64^3	8	256^3	17.9s/frame
Fig. 10	$150 \times 100 \times 100$	96	$600 \times 400 \times 400$	234.48s/frame
Fig. 11	$256 \times 180 \times 180$	441	$1024 \times 720 \times 720$	1046.07s/frame

Details of training runs for different models are listed in the following table. Our standard models that are used unless otherwise indicated are marked with a $^{(*)}$:

Train. iters	data: no. of sims, total frames	training and testing frames	low- high- input res ' res ' tiles	λ_{L_1}	$\lambda_f^{1,\dots,4}$
2D, Fig. 15 a)$^{(*)}$	20, 4000	160, 40	64^2, 256^2, 16^2	5	-10^{-5} for all
2D, Fig. 15 b)	20, 4000	160, 40			$10^{-4}/3$, $-10^{-4}/3$, $-10^{-4}/3$, $10^{-4}/3$
2D, Fig. 17 b)	20, 2400	320, 80			10^{-5} for all
3D, Fig. 9 b)$^{(*)}$	20, 2400	96, 24	64^3, 256^3, 16^3	5	$-10^{-6}/3$ for all
3D, Fig. 16 b)	20, 2400	96, 24			$10^{-6}/3$, $-10^{-6}/3$, $-10^{-6}/3$, $-10^{-6}/3$

Table Cont.	Trainings per step	Training Batch steps ' size	Model weights	Model size (Mb)	Training time (min)
2D, Fig. 15 a)$^{(*)}$	2 for D_s, 2 for D_t, 2 for G	40k, 16	G, 634214 D_s, 706017 D_t, 706529	36.88	798.65
2D, Fig. 15 b)				42.73	905.72
2D, Fig. 17 b)				43.45	877.59
3D, Fig. 9 b)$^{(*)}$	16 for D_s, 16 for D_t, 16 for G	7k, 1	G: 3148014 D_s: 2888161 D_t: 2890209	134.93	12636.22 2 GPUs
3D, Fig. 16 b)				140.79	18231.97 2 GPUs