Latent-space Physics: Towards Learning the Temporal Evolution of Fluid Flow

STEFFEN WIEWEL, Technical University of Munich
MORITZ BECHER, Technical University of Munich
NILS THUEREY, Technical University of Munich

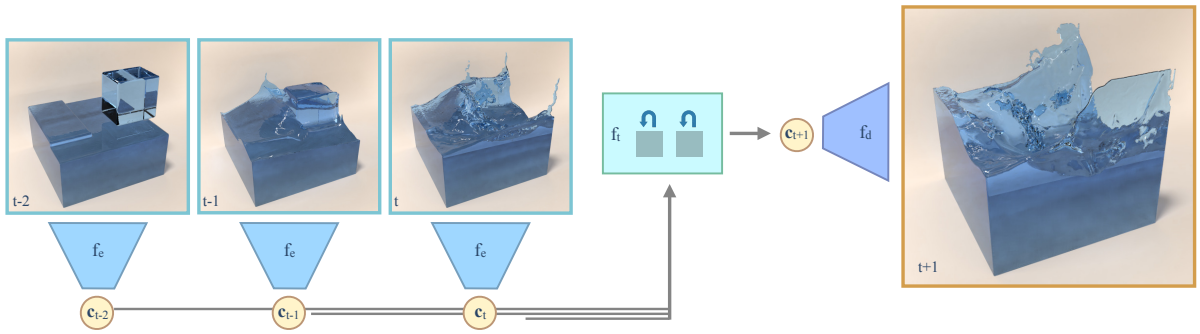


Fig. 1. Our method encodes multiple steps of a simulation field, typically pressure, into a reduced latent representation with a convolutional neural network. A second neural network with LSTM units then predicts the latent space code for one or more future time steps, yielding large reductions in runtime compared to regular solvers.

Our work explores methods for the data-driven inference of temporal evolutions of physical functions with deep learning techniques. More specifically, we target fluid flow problems, and we propose a novel LSTM-based approach to predict the changes of the pressure field over time. The central challenge in this context is the high dimensionality of Eulerian space-time data sets. Key for arriving at a feasible algorithm is a technique for dimensionality reduction based on convolutional neural networks, as well as a special architecture for temporal prediction. We demonstrate that dense 3D+time functions of physics system can be predicted with neural networks, and we arrive at a neural-network based simulation algorithm with significant practical speed-ups. We demonstrate the capabilities of our method with a series of complex liquid simulations, and with a set of single-phase buoyancy simulations. With a set of trained networks, our method is more than two orders of magnitudes faster than a traditional pressure solver. Additionally, we present and discuss a series of detailed evaluations for the different components of our algorithm.

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

Additional Key Words and Phrases: recurrent neural networks, physically-based animation, fluid simulation

ACM Reference Format:

Steffen Wiewel, Moritz Becher, and Nils Thuerey. 2018. Latent-space Physics: Towards Learning the Temporal Evolution of Fluid Flow. 1, 1 (March 2018), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The variables we use to describe real world physical systems typically take the form of complex functions with high dimensionality. Especially for transient numerical simulations, we usually employ continuous models to describe how these functions evolve over time. For such models, the field of computational methods has been highly successful at developing powerful numerical algorithms that accurately and efficiently predict how the natural phenomena under consideration will behave. In the following, we take a different view on this problem: instead of relying on analytic expressions formulated by humans, we use a deep learning approach to infer physical functions based on large collections of input data. More specifically, we will focus on the temporal evolution of complex functions that arise in the context of fluid flows. Fluids encompass a large and important class of materials in human environments, and as such they're particularly interesting candidates for learning models.

While other works have demonstrated that machine learning methods are highly competitive alternatives to traditional methods, e.g.,

This work is supported by the *ERC Starting Grant* 637014.
Authors' addresses: Steffen Wiewel, Technical University of Munich, wiewel@in.tum.de; Moritz Becher, Technical University of Munich, mo.becher@tum.de; Nils Thuerey, Technical University of Munich, nils.thuerey@tum.de.

XXXX-XXXX/2018/3-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

for computing local interactions of particle based liquids [Ladicky et al. 2015], to perform divergence free projections for a single point in time [Tompson et al. 2016; Yang et al. 2016], or to map between coarse and fine discretizations [Chu and Thuerey 2017], few works exist that target temporal evolutions of physical systems. While first works have considered predictions of Lagrangian objects such as rigid bodies [Watters et al. 2017], or relatively simple two-dimensional functions [Farimani et al. 2017], the question whether neural networks (NNs) can predict the evolution of complex three-dimensional functions such as pressure fields of fluids has not previously been addressed. We believe that this is a particularly interesting challenge, as it not only can lead to faster forward simulations, as we will demonstrate below, but also could be useful for giving NNs predictive capabilities for complex inverse problems. E.g., giving models the ability to understand, or at least have an intuition about physics, could help us to analyze video data [Watters et al. 2017] or even control robots [Schenck and Fox 2017].

The complexity of nature at human scales makes it necessary to finely discretize both space and time for traditional numerical methods, in turn leading to a large number of degrees of freedom. Key to our method is reducing the dimensionality of the problem using convolutional neural networks (CNNs) with respect to both time and space. Our method first learns to map the original, three-dimensional problem into a much smaller spatial *latent space*, at the same time learning the inverse mapping. We then train a second network that maps a collection of reduced representations into an encoding of the temporal evolution. This reduced temporal state is then used to output a sequence of spatial latent space representations, which are decoded to yield the full spatial data set for a point in time. A key advantage of CNNs in this context is that they give us a natural way to compute accurate and highly efficient non-linear representations. We will later on demonstrate that the setup for computing this reduced representation strongly influences how well the time network can predict changes over time, and we will demonstrate the generality of our approach with several liquid and single-phase problems. Note that In contrast to other popular techniques for dimensionality reduction, such as methods based on PCA [Treuille et al. 2006], our approach naturally handles complex temporally changing boundary conditions, such as those arising for liquid simulations. The specific contributions of this work are:

- a first LSTM architecture to predict temporal evolutions of dense, physical 3D functions in learned latent spaces,
- an efficient encoder and decoder architecture, which by means of a strong compression, yields a very fast simulation algorithm,
- in addition to a detailed evaluation of architectural choices and parameters.

2 RELATED WORK AND BACKGROUND

Despite being a research topic for a long time [Rumelhart et al. 1988], the interest in neural network algorithms, is a relatively new phenomenon, triggered by seminal works such as *ImageNet* [Krizhevsky et al. 2012]. In computer graphics, such approaches have led to impressive results, e.g., for synthesizing novel viewpoints of natural scenes [Flynn et al. 2016], to generate photorealistic face textures [Saito et al. 2016], and to robustly transfer image styles between photographs [Luan et al. 2017], to name just a few examples. The underlying optimization approximates an unknown function $f^*(x) = y$, by minimizing an associated loss function L such that $f(x, \theta) \approx y$. Here, θ denotes the degrees of freedom of the chosen representation for f. For our algorithm, we will consider deep neural networks. With the right choice of C, e.g., an L_2 norm in the simplest case, such a neural network will approximate the original function f^* as closely as possible given its internal structure. A single layer l of an NN can be written as $a^l = \sigma(W_l a^{l-1} + b_l)$, where a^i is the output of the i'th layer, σ represents an activation function, and W_l, b_l denote weight matrix and bias, respectively. In the following, we collect the weights W_l, b_l for all layers l in θ.

The latent spaces of generative NNs were shown to be powerful tools in image processing and synthesis [Radford et al. 2016; Wu et al. 2016]. They provide a non-linear representation that is closely tied to the given data distribution, and our approach leverages such a latent space to predict the evolution of dense physical functions. While others have demonstrated that trained feature spaces likewise pose very suitable environments for high-level operations with impressive results [Upchurch et al. 2016], we will focus on latent spaces of autoencoder networks in the following. The sequence-to-sequence methods which we will use for time prediction have so far predominantly found applications in the area of natural language processing, e.g., for tasks such as machine translation [Sutskever et al. 2014]. These recurrent networks are especially popular for control tasks in reinforcement learning environments [Mnih et al. 2016]. Recently, impressive results were also achieved for tasks such as automatic video captioning [Xu et al. 2017].

Using neural networks in the context of physics problems is a new area within the field of deep learning methods. Several works have targeted predictions of Lagrangian objects based on image data. E.g., Battaglia et al. [2016] introduced a network architecture to predict two-dimensional physics, that also can be employed for predicting object motions in videos [Watters et al. 2017]. Another line of work proposed a different architecture to encode and predict two-dimensional rigid bodies physics [Chang et al. 2016]. More recently, Ehrhardt et al. use recurrent NNs to predict trajectories and their likelihood for balls in height-field environments [Ehrhardt et al. 2017].

Others targeted predictions of liquid motions for robotic control, with a particular focus on pouring motions [Schenck and Fox 2017]. Farimani et al. [Farimani et al. 2017] recently proposed an adversarial training approach to infer solutions for two-dimensional physics problems, such as heat diffusion and lid driven cavity flows. Other researchers have proposed specialized networks to learn PDEs [Long et al. 2017] by encoding the unknown differential operators with convolutions. While both works share our goal to infer Eulerian functions for physical models, they are limited to relatively simple, two dimensional problems. In contrast, we will demonstrate that our reduced latent space representation can work with complex functions with up to several million degrees of freedom.

We focus on flow physics, for which we employ the well established *Navier-Stokes* (NS) model. Its incompressible form is given by

$$\partial \mathbf{u}/\partial t + \mathbf{u} \cdot \nabla \mathbf{u} = -1/\rho \nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{g}\ , \quad \nabla \cdot \mathbf{u} = 0, \tag{1}$$

where the most important quantities are flow velocity $\mathbf{u}$ and pressure p. The other parameters $\rho, \nu, \mathbf{g}$ denote density, kinematic viscosity and external forces, respectively. For liquids, we will assume that a signed-distance function ϕ is either advected with the flow, or reconstructed from a set of advected particles.

In the area of visual effects, Kass and Miller were the first to employ height field models [Kass and Miller 1990], while Foster and Metaxas employed a first three-dimensional NS solver [Foster and Metaxas 1996]. After Jos Stam proposed an unconditionally stable advection and time integration scheme [1999], it led to powerful liquid solvers based on the particle level-set [Foster and Fedkiw 2001], in conjunction with accurate free surface boundary conditions [Enright et al. 2003]. Since then, the *fluid implicit particle* (FLIP) method has been especially popular for detailed liquid simulations [Zhu and Bridson 2005], and we will use it to generate our training data. Solvers based on these algorithms have subsequently been extended with accurate boundary handling [Batty et al. 2007; Robinson-Mosher et al. 2008], synthetic turbulence [Kim et al. 2008], or narrow band algorithms [Ferstl et al. 2016], to name just a few examples. In addition, an important aspect for many simulations is artistic control [Nielsen et al. 2009; Pan et al. 2013]. A good overview of fluid simulations for computer animation can be found in the book by R. Bridson [2015]. Aiming for more complex systems, other works have targeted coupled reduced order models, or sub-grid coupling effects [Fei et al. 2017; Teng et al. 2016]. While we do not target coupled fluid solvers in our work, these directions of course represent interesting future topics.

Beyond these primarily grid-based techniques, *smoothed particle hydrodynamics* (SPH) are a popular Lagrangian alternative [Macklin et al. 2014; Müller et al. 2003]. However, we will focus on Eulerian solvers in the following, as CNNs are particularly amenable to grid-based discretizations. The pressure function has received special attention, as the underlying iterative solver is often the most expensive component in an algorithm. E.g., techniques for dimensionality reduction [Ando et al. 2015; Lentine et al. 2010], and fast solvers [Ihmsen et al. 2014; McAdams et al. 2010] have been proposed to diminish its runtime impact.

In the context of fluid simulations and machine learning for animation, a regression forest based approach for learning SPH interactions has been proposed by Ladicky et al. [2015]. Other graphics works have targeted learning flow descriptors with CNNs [Chu and Thuerey 2017], or learning the statistics of splash formation [Um et al. 2017]. While pressure projection algorithms with CNNs [Tompson et al. 2016; Yang et al. 2016] might seem similar to our work on first sight, they are largely orthogonal. Instead of targeting divergence freeness for a single instance in time, our work aims for learning its temporal evolution over the course of many time steps. An important difference is also that the CNN-based projection so far has only been demonstrated for smoke simulations, similar to other model-reduced simulation approaches [Treuille et al. 2006]. For all of these methods, handling the strongly varying free surface boundary conditions remains an open challenge, and we demonstrate below that our approach works especially well for liquids.

3 METHOD

The central goal of our method is to predict future states of a physical function $\mathbf{x}$. While previous works often consider low dimensional Lagrangian states such as center of mass positions, $\mathbf{x}$ takes the form of a dense Eulerian function in our setting. E.g., it can represent a spatio-temporal pressure function, or velocity field. Thus, we consider $\mathbf{x} : \mathbb{R}^3 \times \mathbb{R}^+ \rightarrow \mathbb{R}^d$, with $d = 1$ for scalar functions such as pressure, and $d = 3$ for vectorial functions such as velocity fields. As a consequence, $\mathbf{x}$ has high dimensionality when discretized, typically on the order of millions of spatial degrees of freedom.

Given a set of parameters θ and a function representation f_t, which takes the form of a trained neural network model here, our goal is to predict the future state $\mathbf{x}(t+h)$ as closely as possible given a current state and a series of n previous states, i.e.,

$$f_t\left(\mathbf{x}(t-nh), ..., \mathbf{x}(t-h), \mathbf{x}(t)\right) \approx \mathbf{x}(t+h). \tag{2}$$

Due to the high dimensionality of $\mathbf{x}$, trying to solve this problem in this way would be extremely costly. Thus, we employ two additional functions f_d and f_e, that compute a low dimensional encoding. f_e maps into an m_s dimensional space $\mathbf{c} \in \mathbb{R}^{m_s}$ with $f_e(\mathbf{x}(t)) = \mathbf{c}$. Both functions are represented by trained neural networks, which are trained such that $f_d(f_e(\mathbf{x}(t))) \approx \mathbf{x}(t)$. Thus, f_d and f_e here represent spatial decoder and encoder models, respectively. Given such an en- and decoder, we rephrase the problem above as

$$f_d\Big(f_t\left(f_e(\mathbf{x}(t-nh)), ..., f_e(\mathbf{x}(t-h)), f_e(\mathbf{x}(t))\right)\Big) \approx \mathbf{x}(t+h). \tag{3}$$

As we will also use CNNs for f_d and f_e, the space of $\mathbf{c}$ is given by the *latent space* of a neural network, and we choose its dimensionality m such that the temporal prediction problem of Eq. (3) becomes feasible for dense three dimensional samples. Our time network will likewise employ an encoder-decoder structure, and turn the stack of encoded data sets $f_e(\mathbf{x})$ into a reduced representation $\mathbf{d}$, which we will call time *context* below. Despite, the similar high-level structure, the time network will require a significantly different setup compared to its spatial counterpart, as we will detail in Sec. 3.2.

3.1 Reducing Dimensionality

In order to reduce the spatial dimensionality of our inference problem, we employ a fully convolutional autoencoder architecture. Autoencoders represent a popular class of unsupervised architectures for learning reduced encodings. Our autoencoder consists of the aforementioned encoding and decoding functions f_e, f_d, and its goal is to reconstruct the quantity $\mathbf{x}$ as accurately as possible w.r.t. an L_2 norm:

$$\min_{\theta_d, \theta_e} |f_d(f_e(\mathbf{x}(t))) - \mathbf{x}(t)|_2. \tag{4}$$

Here θ_d, θ_e denote the parameters of decoder and encoder, respectively. To exploit the benefits of depth in neural networks [Hinton and Salakhutdinov 2006], we use a series of convolutional layers

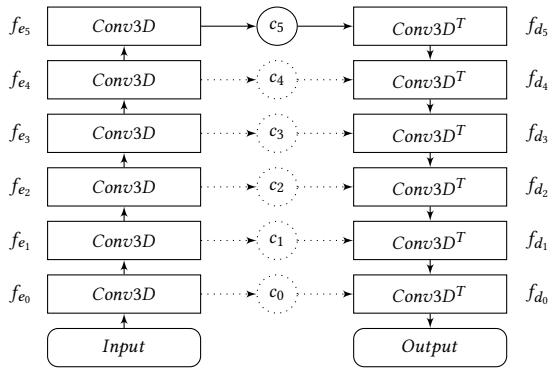


Fig. 2. Overview of the autoencoder network architecture
See Table 1 for a detailed description of the layers. Note that the paths from f_{e_k} over c_k to f_{d_k} are only active in pretraining stage k. After pretraining only the path f_{e_5} over c_5 to f_{d_5} remains active.

Layer	Kernel	Stride	Activation	Output	Features
Input				$\mathbf{r}/1$	d
f_{e_0}	4	2	Linear	$\mathbf{r}/2$	32
f_{e_1}	2	2	LeakyReLU	$\mathbf{r}/4$	64
f_{e_2}	2	2	LeakyReLU	$\mathbf{r}/8$	128
f_{e_3}	2	2	LeakyReLU	$\mathbf{r}/16$	256
f_{e_4}	2	2	LeakyReLU	$\mathbf{r}/32$	512
f_{e_5}	2	2	LeakyReLU	$\mathbf{r}/64$	1024
c_5				$\mathbf{r}/64$	1024
f_{d_5}	2	2	LeakyReLU	$\mathbf{r}/32$	512
f_{d_4}	2	2	LeakyReLU	$\mathbf{r}/16$	256
f_{d_3}	2	2	LeakyReLU	$\mathbf{r}/8$	128
f_{d_2}	2	2	LeakyReLU	$\mathbf{r}/4$	64
f_{d_1}	2	2	LeakyReLU	$\mathbf{r}/2$	32
f_{d_0}	4	2	Linear	$\mathbf{r}/1$	d

Table 1. Parameters of the autoencoder layers
$\mathbf{r} \in \mathbb{N}^3$ denotes the resolution of the data, and $d \in \mathbb{N}$ its dimensionality.

activated by leaky rectified linear units (LeakyReLU) [Maas et al. 2013] for encoder and decoder to create a deep convolutional autoencoder [Masci et al. 2011]. As this network architecture forces all data through a bottleneck of dimensionality m_s, the autoencoder is forced to learn to represent all crucial properties of the data with only m_s values. This yields the reduced latent space that later on will be used to efficiently compute the temporal evolution.

Note that in contrast to the field of natural images [Simonyan and Zisserman 2014], currently no pre-trained models exist that are suitable for physics problems. In order to work with the large Eulerian data sets of fluid simulations, we found *greedy pre-training* to be a crucial tool for speeding up the training process. We denote layers in the encoder and decoder stack as f_{e_i} and f_{d_j}, where $i, j \in [0, n]$ denote the depth from the input and output layers, n being the depth of the latent space layer. In our network architecture, encoder and decoder layers with $i = j$ have to match, i.e., the output shape of f_{e_i} has to be identical to that of f_{d_j} and vice versa. This setup allows for a greedy, layer-wise pretraining of the autoencoder, as proposed by Bengio et al. [Bengio et al. 2007], where beginning from a shallow single layer deep autoencoder, additional layers are added to the model forming a series of deeper models for each stage. The optimization problem of such a stacked autoencoder in pretraining is therefore formulated as

$$\min_{\theta_{e_{0\dots k}}, \theta_{d_{0\dots k}}} |f_{d_0} \circ f_{d_1} \circ \dots \circ f_{d_k}(f_{e_k} \circ f_{e_{k-1}} \circ \dots \circ f_{e_0}(\mathbf{x}(t))) - \mathbf{x}(t)|_2, \quad (5)$$

with $\theta_{e_{0\dots k}}, \theta_{d_{0\dots k}}$ denoting the parameters of the sub-stack for pretraining stage k, and $\circ$ denoting composition of functions. For our final models, we typically use a depth $n = 5$, and thus perform 6 runs of pretraining before training the complete model.

In addition, our autoencoder does not use any pooling layers, but instead only relies on strided convolutional layers. This means we apply convolutions with a stride of s, skipping $s-1$ entries when applying the convolutional kernel. We assume the input is padded, and hence for $s = 1$ the output size matches the input, while choosing a stride $s > 1$ results in a downsampled output [**?**]. Equivalently the decoder network employs strided transposed convolutions, where strided application increases the output dimensions by a factor of s. The details of the network architecture, with corresponding strides and kernel sizes can be found in Table 1.

When using autoencoders as generative models, it is often preferable to obtain a meaningful and sparse latent space representation. Here, *variational* autoencoders (VAEs) are a particularly popular choice, which learn a normalization of the range of all latent space dimensions [Rezende et al. 2014], in addition to the encoding itself. We have tested a VAE instead of a regular autoencoder, but found that the resulting normalized latent space was less suitable for temporal inference. We will present these results in more detail below.

Overall, the autoencoder provides us with a reduced, non-linear representation of the input data. The inherent non-linearity is a particularly important aspect here. Previous works have thoroughly demonstrated the advantages over linear bases such as those computed by PCA [Bourlard and Kamp 1988], and the neural network latent space as basis for our algorithm is one of the central differences to previous work [Treuille et al. 2006; Wicke et al. 2009].

3.2 Prediction of Future States

In contrast to the spatial reduction network above, which receives the full spatial input at once and infers a latent space coordinate without any data internal to the network, the time network uses a recurrent architecture for predicting the evolution over time. More specifically, we propose using a *sequence-to-sequence* network [Sutskever et al. 2014], which is a popular architecture for natural language processing tasks like machine translation or the synthesis of handwriting [Cho et al. 2014; Graves 2013]. It receives a series of inputs one by one, and computes its output iteratively with the help of an internal network state.

To achieve this, the regular neural network nodes are replaced by more complex, gated units, the most popular of which are the long short-term memory (LSTM) [Hochreiter and Schmidhuber 1997] unit and the gated recurrent unit [Cho et al. 2014]. Our networks

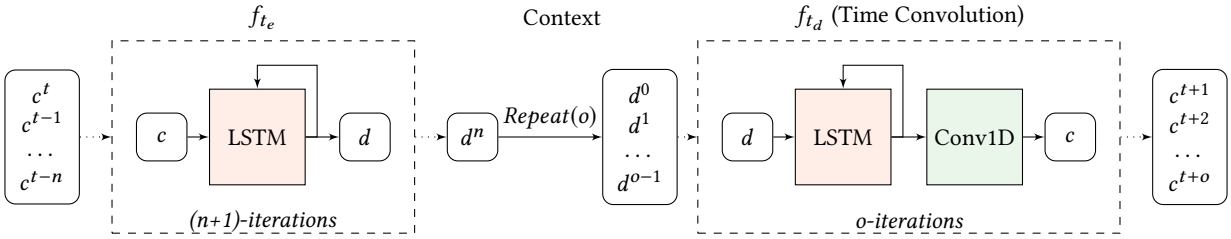


Fig. 3. Sequence to Sequence Network
The dashed boxes indicate an iterative evaluation. As an alternative architecture, the decoder part (right side) can be replaced with additional LSTM layers, as shown in Fig. 4. See Table 2 for details of the layers.

will be based on the former, and we summarize the internals of LSTM units in Appendix A. Note that in contrast to the spatial reduction, which very heavily relies on convolutions, we cannot employ similar convolutions for the time data sets. While it is a valid assumption that each entry of a latent space vector $\mathbf{c}$ varies smoothly in time, the order of the entries is arbitrary and we cannot make any assumptions about local neighborhoods within $\mathbf{c}$. As such, convolving $\mathbf{c}$ with a filter along the latent space entries typically does not give meaningful results. Instead, our time network will use convolutions along the time axis in selected places, and primarily rely on fully connected layers of LSTM units.

In the following we will use the short form $\mathbf{c}^t = f_e(\mathbf{x}(t))$ to denote a spatial latent space point at time t. The prediction network transforms a sequence of $n+1$ chronological, encoded input states $X = (\mathbf{c}^{t-nh}, \dots, \mathbf{c}^{t-h}, \mathbf{c}^t)$ into a consecutive list of o predicted future states $Y = (\mathbf{c}^{t+h}, \dots, \mathbf{c}^{t+oh})$. The minimization problem solved during training thus aims for minimizing the mean absolute error between the o predicted and ground truth states with an L_1 norm:

$$\min_{\theta_t} \left| f_t\left(\mathbf{c}^{t-nh}, \dots, \mathbf{c}^{t-h}, \mathbf{c}^t\right) - \left[\mathbf{c}^{t+h}, \dots, \mathbf{c}^{t+oh}\right] \right|_1 . \qquad (6)$$

Here θ_t denotes the parameters of the prediction network, and $[\cdot,\cdot]$ denotes concatenation of the $\mathbf{c}$ vectors.

The network approximates the desired function with the help of an internal temporal context with a dimension (m_t), which we will denote as $\mathbf{d}$. Thus, the first part of our time network represents a recurrent encoder module, transforming $n+1$ latent space points into a time context $\mathbf{d}$. The time decoder module has a similar structure, and is likewise realized with layers of LSTM units. This module takes a context $\mathbf{d}$ as input, and outputs a series of future, spatial latent space representations. By means of its internal state, the time decoder is trained to predict o future states when receiving the same context $\mathbf{d}$ repeatedly. We use tanh activations for all LSTM layers, and hard sigmoid functions for efficient, internal activations.

Note that the iterative nature is shared by encoder and decoder module of the time network. I.e., the encoder actually internally produces $n+1$ contexts, the first n of which are intermediate contexts. These intermediate contexts are only required for the feedback loop internal to the corresponding LSTM layer, and are discarded afterwards. We only keep the very last context in order to pass it to the decoder part of the network. This context is repeated o times, in order for the decoder LSTM to infer the desired future states.

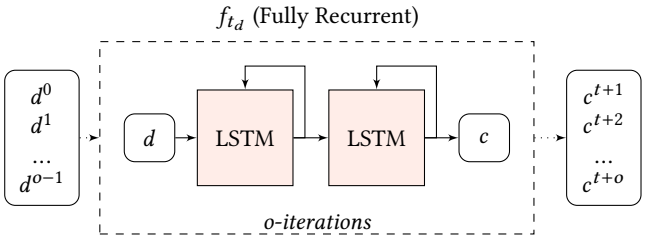


Fig. 4. Alternative Sequence to Sequence Decoder (Fully Recurrent)

	Layer (Type)	Activation	Output Shape
f_{t_e}	Input		$(n+1, m_s)$
	LSTM	tanh	(m_t)
Context	Repeat		(o, m_t)
f_{t_d}	LSTM	tanh	(o, m_{t_d})
	Conv1D	linear	(o, m_s)

Table 2. Recurrent encoder with time convolutional decoder architecture (Fig. 3)

While the autoencoder, thanks to its fully convolutional architecture, could be applied to inputs of varying size, the prediction network is trained for fixed latent space inputs, and internal context sizes. Correspondingly, when the latent space size m_s changes, it influences the time network size as shown in Table 3. While this clearly means that the time network has to be re-trained from scratch when the latent space size is changed, we have found this not to be critical in practice. The time network takes significantly less time to train than the space network, as we will discuss in more detail in Sec. 5.

3.3 Dimensionality and Network Size

A central challenge for deep learning problems involving fluid flow is the large number of degrees of freedom present in three-dimensional data sets. This quickly leads to layers with large numbers of nodes

	Layer (Type)	Activation	Output Shape
f_{t_e}	Input		$(n+1, m_s)$
	LSTM	tanh	(m_t)
Context	Repeat		(o, m_t)
f_{t_d}	LSTM	tanh	(o, m_{t_d})
	LSTM	tanh	(o, m_s)

Table 3. Fully recurrent network architecture (Fig. 4)

– from hundreds to thousands per layer. Here, a potentially unexpected side effect of using LSTM nodes is the number of weights they require. The local feedback loops for the gates of an LSTM unit all have trainable weights, and as such induce an $n \times n$ weight matrix for n LSTM units. E.g., even for a simple network with a one dimensional input and output, and a single hidden layer of 1000 LSTM units, with only 2×1000 connections and weights between in-, output and middle layer, the LSTM layer internally stores 1000^2 weights for its temporal feedback loop. In practice, LSTM units have *input, forget and output* gates in addition to the feedback connections, leading to $4n^2$ internal weights for an LSTM layer of size n. Details can be found in Appendix A.

Keeping the number of weights at a minimum is in general extremely important to prevent overfitting, reduce execution times, and to arrive at networks which are able to generalize. To prevent the number of weights from exploding due to large LSTM layers, we propose the mixed use of LSTM units and 1D convolutions for our final temporal network architecture. Here, we change the decoder part of the network to consist of a single dense LSTM layer that generates a sequence of o vectors of size m_{t_d}. Instead of processing these vectors with another dense LSTM layer as before, we concatenate the outputs into a single tensor, and employ a single one-dimensional convolution translating the intermediate vector dimension into the required m_s dimension for the latent space. Thus, the 1D convolution works along the vector content, and is applied in the same way to all o outputs. Unlike the dense LSTM layers, the 1D convolution does not have a quadratic weight footprint, and purely depend on the size of input and output vectors. This architecture is summarized in Table 2.

4 FLUID DATA

We rely on a NS solver with operator splitting to give us approximations at discrete points in space and time. On a high level, the solver contains the following steps: computing motion of the fluid with the help of advection steps for the velocity, evaluating external forces, and then computing the pressure field [Bridson 2015]. In addition, a visible, passive quantity such as smoke density ρ, or a level-set representation ϕ for free surface flows could be advected in parallel to the velocity itself. Calculating the pressure typically involves solving a large system of equations, and the gradient of the resulting pressure is used to make the flow divergence free. Depending on the choice of physical quantity to infer with our framework, different simulation algorithms emerge.

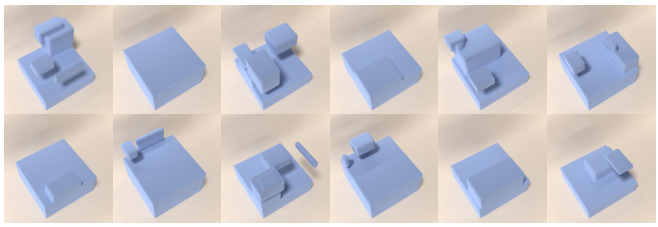


Fig. 5. Examples of initial scene states in the liquid64 data set.

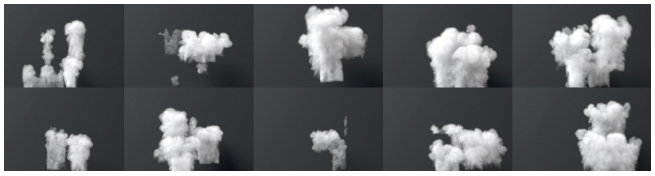


Fig. 6. Examples states of the smoke128 data set after 65 steps.

The central quantities of a fluid solve are flow velocity $\mathbf{u}$, pressure p, and potentially visible quantities such as ϕ and ρ. If not otherwise noted, we use our prediction network during each simulation step, and use the inferred, decoded value instead of executing the corresponding numerical approximation step. When targeting $\mathbf{u}$ with our method, this means that we can omit velocity advection as well as pressure solve, while the inference of p means that we only omit the pressure solve, but still need to perform advection and velocity correction with the pressure gradient. While the latter requires more computations, the pressure solve is typically the most time consuming part with a super-linear complexity, and as such both options have comparable runtimes.

A variant for all of these cases is to only rely on the network prediction for a limited time interval of i_p time steps, and then perform a single full simulation step. I.e., for $i_p = 0$ the network is not used at all, while $i_p = \infty$ is identical to the full network prediction described in the previous paragraph. We will investigate intervals on the order of 4 to 14 steps. This simulation variant represents a numerical time integration and network prediction hybrid, that can have advantages to prevent drift from the network predictions. We will denote this version as an *interval prediction* below.

We leave inferring advected quantities such as smoke density ρ or surface level-set ϕ for future work, as this goal would lead to substantially different approaches. Previous works have attempted generating advected quantities directly, e.g., by using space-time deformations [Prantl et al. 2017; Raveendran et al. 2014], but we will focus on simulation algorithms that explicitly compute transport processes with advection algorithms in the following.

4.1 Free Surface Boundary Conditions

While re-computing the advected quantities ensures temporal coherence, it causes difficulties when the state of an advected system does not match the state of the input data anymore. As the training data relies on specific motions produced by a numerical solver, any

deviations from this solution can cause, e.g., the liquid surface, to have a different position than the one it had when the inputs were calculated. This could happen due to numerical drift, but especially the compression of the autoencoder typically leads to differences in the solutions.

As a consequence for liquids, we need to ensure that the zero pressure boundary condition $p = 0$ is accurately enforced at the current position of the liquid surface. Hence, we employ the boundary condition alignment step introduced by Ando et al. [2015]. They run into similar difficulties due to a coarse pressure projection step that not necessarily yields an exact solution at the surface. As suggested there, we perform 2-3 iterations of Jacobi steps for the pressure Poisson problem in a narrow band around the surface. This suffices to correct the free surface boundary conditions such that they match the liquid surface, before applying the pressure gradient to the flow field.

4.2 Pressure Splitting

Although our final version will directly infer the full pressure field, we have experimented with a pressure splitting approach, under the hypothesis that this could simplify the inference problem for the LSTM. Assuming a fluid at rest on a ground with height z_g, a hydrostatic pressure p_s for cell at height z, can be calculated as $p_s(z) = p(z_0) + \frac{1}{A}\int_z^{z_0}\iint_A \mathbf{g}\rho(h)\,\mathrm{d}x\mathrm{d}y\,\mathrm{d}h$, with z_0, p_0, A denoting surface height, surface pressure, and cell area, respectively. As density and gravity can be treated as constant in our setting, this further simplifies to $p_s = \rho\mathbf{g}(z - z_0)$. While this form can be evaluated very efficiently, it has the drawback that it is only valid for fluids in hydrostatic equilibrium, and typically cannot be used for dynamic simulations in 3D.

Given a data-driven method to predict pressure fields, we can incorporate the hydrostatic pressure into a 3D liquid simulation by decomposing the regular pressure field into hydrostatic and dynamic components $p = p_s + p_d$, such that our autoencoder separately receives and encodes the two fields p_s and p_d. With this split pressure, the autoencoder could potentially put more emphasis on the small scale fluctuations p_d from the hydrostatic pressure gradient. We will evaluate this option as an alternative approach in more detail below.

4.3 Data Sets

For generating the sets of training data, variance in the geometric layout of the scenes was ensured by randomizing shape and appearance of various types of fluid volumes. We target scenes with high complexity, i.e., strong visible splashes and vortices, and large CFL numbers significantly above 1 (typically around 2-3). For each of our data sets, we generate n_s scenes of different initial conditions, for which we discard the first n_w time steps as warm-up phase. Due to initialization, these are typically smooth and regular, and thus less representative of the phenomena we are interested in. After warm-up, we store a fixed number of n_t time steps as training data, resulting in a final size of $n_s n_t$ full spatial data sets. Each data set content is normalized to the range of [-1,1].

	liquid64	liquid128	smoke128
Scenes	4000	800	800
Time steps	100	100	100
Size	419.43GB	671.09GB	671.09GB
Size f_e	1.64GB	2.62GB	2.62GB
Compression ratio	256	256	256

Table 4. List of the augmented data sets for the total pressure architecture. Compression is achieved by the encoder part of the autoencoder f_e.

We have tested our approach with three different 3D data sets, two containing liquids, and an additional one for smoke simulations. The liquid data sets with spatial resolutions of 64^3 and 128^3 contain resting liquid in a cubic container, with a liquid height varying in an interval of 30-60% of the domain height. We then add a standing column of randomized size to induce sloshing, and up to 4 randomly placed liquid volumes with randomized initial velocities as falling drops. Examples of initial states can be found in Fig. 5. The phenomena covered by these scenes therefore range from resting liquids with smooth surface waves, to strong splashes, as well as falling and colliding bodies of liquid. In the following, we will denote these data sets as *liquid64* and *liquid128*, respectively. For the final trainings an augmented version of the data set was used which mirrored the data sets along XY and YZ, leading to the data set sizes given in Table 4.

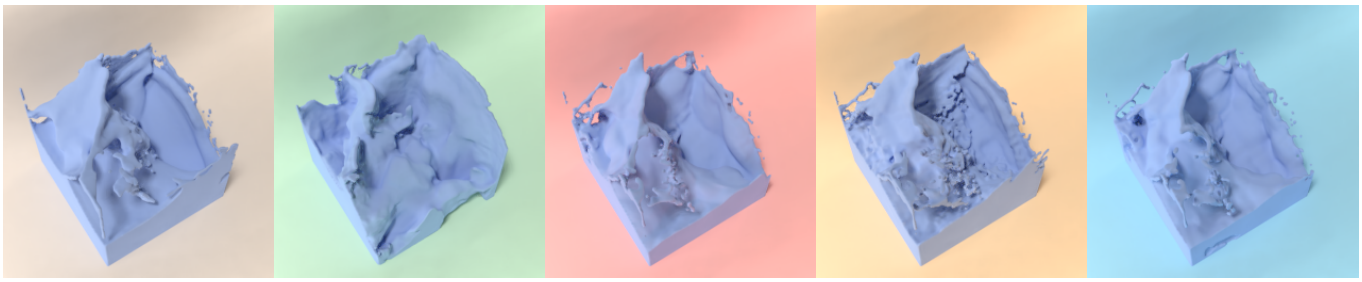
In addition, we consider a data set containing single-phase flows with buoyant smoke based on the Boussinesq model. We place 4 to 10 smoke inflow regions into an empty domain at rest, and then simulate the rising plumes of hot smoke that develop over time in different configurations. Examples are shown in Fig. 6. This data set contains substantially different fluid phenomena than the other two liquid data sets, and serves as a test for the generalizing capabilities of our approach. We will denote this data set as *smoke128* below.

5 EVALUATION AND TRAINING

In the following we will evaluate the different options discussed in the previous section with respect to their prediction accuracies. In particular, we will compare different physical quantities as the basis for our predictions. In terms of evaluation metrics, L_2 errors are typically insufficient due to the inherent averaging. Instead, we will employ a metric based on mean Hausdorff-distances between ground truth and predicted liquid surfaces.

Similar metrics are popular, e.g., for comparing images and for shape retrieval [Huttenlocher et al. 1993; Li et al. 2015]. Given two signed distance functions $\phi_r, \phi_p : \mathbb{R}^3 \rightarrow \mathbb{R}$ representing reference and predicted surfaces as their zero level sets $S_{r,p} = \{x_i | \phi_{r,p}(x_i) = 0\}$, we compute the error metric e_h as:

$$e_h = \max\left(\frac{1}{|S_p|}\sum_{\mathbf{p}_1 \in S_p}\phi_r(\mathbf{p}_1), \frac{1}{|S_r|}\sum_{\mathbf{p}_2 \in S_r}\phi_p(\mathbf{p}_2)\right), \tag{7}$$



(a) Reference (b) Velocity (c) Variational split pressure (d) Total pressure (e) Split pressure

Fig. 7. Renderings for different quantities encoded and decoded with an autoencoder trained for the indicated quantity. For time integration, these simulations rely on numerical methods. The image on the left shows a regular simulation for reference.

as even this surface distance metric has its shortcomings in certain situations, we will furthermore show and discuss visual details and differences between the different approaches in Sec. 6. Unless otherwise noted, we start measuring differences for ten test scenes after 50 time steps, and the following comparisons use the *liquid64* data set.

5.1 Spatial Encoding

As our prediction model can operate on different physical quantities $\mathbf{x}$, we first evaluate the accuracy of our autoencoder when it is used to reconstruct the different quantities. For these tests, we only perform a spatial encoding and decoding without any temporal prediction. Thus, at the end of a fluid solving time step, we encode the variable under consideration $\mathbf{c} = f_e(\mathbf{x})$, and then restore it from its latent space representation $\mathbf{x}' = f_d(\mathbf{c})$. The simulation then continues with the new value $\mathbf{x}'$. This test serves as a baseline test for accuracy, to measure how well the spatial encoding performs in conjunction with a numerical time integration scheme. In the following, we will compare flow velocity $\mathbf{u}$, total pressure p, and split pressure $[p_s, p_d]$. For the liquid64 data set we use a latent space size of $m_s = 1024$. We train a new autoencoder for each quantity, and we additionally consider a variational autoencoder for the split pressure. Training times for the autoencoders were two days on average, including pre-training. Note that $\mathbf{u}$ is a vector quantity, and similarly, split pressure contains two inferred values per cell. To train the different autoencoders, we use 6 epochs of pretraining and 25 epochs of training using an Adam optimizer, with a learning rate of 0.001 and L2 regularization with strength 0.005. For training we used 80% of the data set, 10% for validation during training, and another 10% for testing. When using the liquid64 data set, one epoch of training therefore consists of 80000 samples.[1]

In Fig. 8a the level-set errors computed with Eq. (7) for the different simulations quantities, each averaged over ten different simulation tests, are shown over time. The sharp increase of the error for the velocity autoencoder indicates that it is not a suitable choice for our latent space approach, as we will additionally demonstrate later on in conjunction with the time prediction. All three pressure versions, on the other hand, exhibit significantly lower errors. In this setting, the total pressure version clearly has the lowest error with respect to the averaged surface distance.

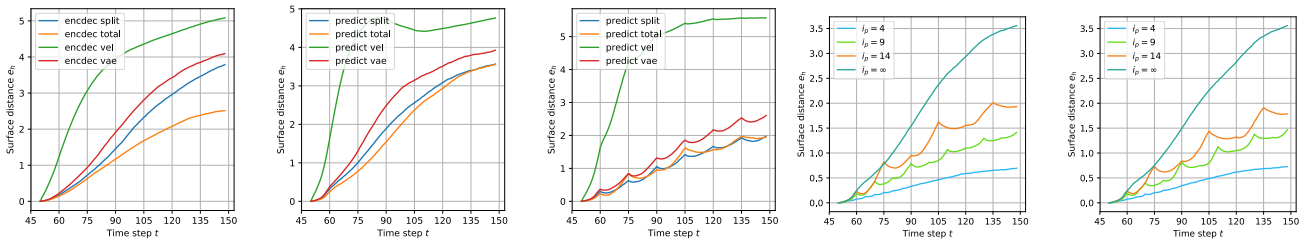
Rendered surfaces for selected frames can be found in Fig. 7. All surfaces are the result of a long series of consecutive latent space encodings, 50 steps in this example, in order to clearly show where the autoencoder introduces compression artifacts. The large differences in surface positions for the velocity autoencoder coincide with the high surface error measurements. It yields smooth results, but clearly fails to recover the overall surface shape. Differences between the three pressure-based approaches are less obvious, as all three recover the large scale sheets and splashes of the flow. It is apparent that the split pressure does not translate into an easier task for the autoencoder, and especially the VAE introduces significant artifacts near the liquid surface.

5.2 Temporal Prediction

Next, we evaluate reconstruction quality when including the temporal prediction network. Thus, now a quantity $\mathbf{x}'$ is inferred based on a series of previous latent space points, and used in the fluid simulation loop. For the following tests, our prediction model uses a history of 6, and infers the next time step, thus $o = 1$. For a resolution of 64^3 the fully recurrent network (compare Table 3) contains $m_t = 700$ LSTM units in the time encoder f_{t_e}, and $m_{t_d} = 1500$ in the time decoder f_{t_d}. The dimensionality of the output is given by the latent-space of the autoencoder, which is $m_s = 1024$ in this case.

A dropout rate of $1.32 \cdot 10^{-2}$ with a recurrent dropout of 0.385, and a learning rate of $1.26 \cdot 10^{-4}$ with a decay factor of $3.34 \cdot 10^{-4}$ were used for all trainings of the prediction network. All prediction models converged after training for 50 epochs with RMSProp. Each epoch traverses the training part of the full data set with augmentation, containing 85% of the entries. Those entries cover 340000 time steps, resulting in 319600 training samples for the liquid64 case. A training run for our LSTM took 2 hours, on average. The hyperparameters were found by empirically sampling their parameter domains, and cross checking their influence on other hyperparameters. See Appendix B for further information on the hyperparameter search process, and the resulting changes of the training loss.

[1]For training the AE with liquid64 we did not find it necessary to activate data augmentation.



(a) Different quantities using only our autoencoder.

(b) Different quantities using our full framework with $i_p = \infty$.

(c) Different quantities using our full framework with $i_p = 14$.

(d) Different prediction intervals i_p for p_t with our full framework.

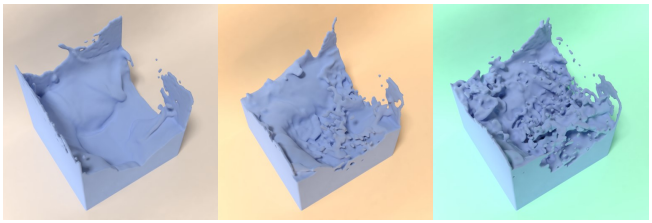
(e) Prediction intervals i_p for p_t with our full framework for $o = 3$.

Fig. 8. Surface distance errors averaged over ten *liquid64 simulations*.

We evaluate the surface error for the same set of ten simulation cases, but instead of relying on a numerical time integration, we use only our LSTM network to predict the temporal evolution of the system. The prediction is again started after 50 steps. Our LSTM receives the last 6 of these, and is then used to predict the next time step. For subsequent predictions, the network receives its own previous output to infer a future state. The corresponding level-set errors over time are shown in Fig. 8b, and a visual comparison can be found in Fig. 14. The error measurements clearly show that using the velocity leads to overly large errors. The error is low for all three pressure variants, especially in comparison to the traditional time integration of Fig. 8a. Thus, given the compression of an autoencoder, our time network does very well at predicting the future states based on a sequence of latent space values. We also evaluate the interval prediction described in Sec. 4. Here we employ the LSTM for i_p consecutive steps, and then perform a single regular simulation step. The results for $i_p = 14$ are shown in Fig. 8c. While the interval prediction barely influences the velocity version, it significantly reduces the surface error for the pressure variants. As velocities are incrementally updated for the pressure inference, the full projections in intervals significantly improve the ability of the LSTM to predict future states.

As our solve targets divergence, we also measured how well the predicted pressure fields enforce divergence freeness over time. As a baseline, the numerical solver led to a residual divergence of $3.1 \cdot 10^{-3}$ on average. In contrast, the pressure field predicted by our LSTM on average introduced a $2.1 \cdot 10^{-4}$ increase of divergence per time step. Thus, the per time step error is well below the accuracy of our reference solver, and especially in combination with the interval predictions, we did not notice any significant changes in mass conservation compared to the reference simulations.

We also compare a fully recurrent LSTM with our proposed time convolutional alternative. The resulting predictions for an example scene, in comparison to the ground truth simulation, can be found in Fig. 9. In this scene, representative for our other test runs, the time convolution outperforms the fully recurrent version in terms of accuracy. In addition, it features the significantly lower weight count, as outlined above. Thus, in conclusion, the total pressure in combination with the time-convolutional LSTM network with $o = 1$ will be used as our "standard" version in the following.



Reference | Time Convolution | Fully Recurrent

Fig. 9. This image compares a time-convolutional and a fully recurrent prediction (both for $i_p = 14$), with the ground truth result on the left. The time convolution clearly outperforms the fully recurrent architecture.

6 RESULTS

We now apply our model to the additional, larger data sets, and we will highlight the performance in more detail. First, we demonstrate how our method performs on the liquid128 data set, with its eight times larger number of degrees of freedom per volume. Correspondingly, we use a latent space size of $m_s = 8192$, and the time convolutional prediction network (compare Table 2) contains $m_t = 1000$ LSTM units in the time encoder f_{t_e} and $m_{t_d} = 1500$ in the time decoder f_{t_d}.

Despite the additional complexity of this data set, our method leads to robust temporal predictions. The large time steps in this setting requires significant per time-step updates, the large-scale motions of the liquid are accurately predicted by our method. However, especially for interval predictions with larger intervals, small scale errors can accumulate, and lead to splash formations that differ from the reference simulation. Fig. 11 and Fig. 12 show more realistically rendered simulations for $i_p = 4$. To verify that our method allows simulation to come to a rest, we have simulated a scene for 650 time steps, shown in Fig. 10. Corresponding animation of this setup, and for various additional scenes can be found in the accompanying video.

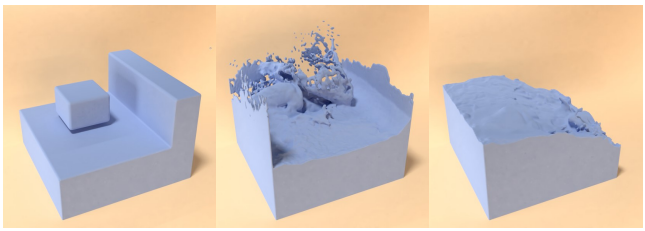
We have experimented with applying our model to smoke scenes. Several example predictions of buoyant smoke with resolutions of 128^3 can be seen in Fig. 13. While our model can predict the evolution and motion of the vortex cores that dominate these scenes, we noticed a tendency to underestimate pressure values, and to reduce small-scale motions. Thus, while our model successfully captures a significant part of the underlying physics, there is a clear room for improvement. E.g., we expect that providing the network with information about the position of smoke in the scene could improve the inference.

We also evaluate how our model compares to the baseline model when multiple output time steps are inferred at once by the decoder layer of our LSTM. In Fig. 8e a measurement of the surface distances for the liquid64 data set with $o = 3$ can be found. Thus, our model predicts three future pressure fields from a single time context $\mathbf{d}$. Comparing the resulting error measurements with the single step prediction of Fig. 8d, it is visible that the accuracy barely degrades when multiple steps are predicted. However, this case is significantly more efficient for our model. The corresponding measurements can be found in Table 5. The $o = 3$ prediction only requires 30% more time to evaluate, despite generating three times as many predictions. In our video, it is also demonstrated that this version is visually very close to the single time step version.

Overall, our method leads to significant speed-ups compared to regular pressure solvers, and our method especially pays off when targeting larger volumes. For the resolution of 128^3, our model predicts the pressure fields several orders of magnitude faster than the CPU version, including encoding and decoding stages. The latter take 4.1ms and 3.3ms [2], respectively. Evaluating the trained LSTM network itself is more expensive, but still very fast with 9.5ms, on average. In total, this execution is ca. 155× faster than our CPU-based MIC-CG pressure solver [Bridson 2015], running with eight threads on a modern CPU. Even when taking into account a factor of ca. 10× for GPUs due to their better memory bandwidth, this leaves a speedup by a factor of more than 15×, pointing to a significant gain in efficiency for our LSTM-based prediction. Due to the super-linear complexity of most iterative solvers, we expect even larger gains for increased resolutions.

Combining the GPU-based prediction with a regular, CPU-based solver in a naive way, we can already achieve practical speedups. This case is clearly sub-optimal for our method, as a significant amount of data has to be transferred back and forth for every time step, for a very short calculation on the GPU. With this simple, serial implementation, we already achieve practical speed-ups of 10x for an interval prediction with $i_p = 14$. Detailed performance measurements can be found in Table 6. For these measurements, we have not optimized data-transfer or scheduling of the GPU execution, and simply rely on a python-based tensorflow call, and hence there is significant room for optimization.

[2]Measured with the *tensorflow timeline* tools on Intel i7 6700k (4GHz) and Nvidia Geforce GTX970.



(a) $t = 0$ (b) $t = 300$ (c) $t = 600$

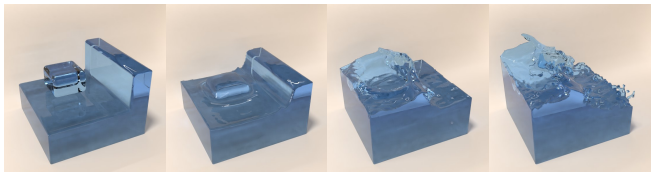
Fig. 10. Renderings of a long running prediction scene for the liquid128 data set with $i_p = 4$. The fluid successfully comes to rest at the end of the simulation.

	Solve		Speedup
Reference	169ms		1.0
	Enc/Dec	**Prediction**	
Core exec., $o = 1$	3.8ms + 3.2ms	9.6ms	10.2
Core exec., $o = 3$	3.9ms + 3 ∗ 3.3ms	12.5ms	19.3

Table 5. Performance measurement of ten liquid64 example scenes over 150 simulation steps each[2].

Interval i_p	Solve	Mean surf. dist	Speedup
Reference	2.629s	0.0	1.0
4	0.600s	0.0187	4.4
9	0.335s	0.0300	7.8
14	0.244s	0.0365	10.1
∞	0.047s	0.0479	55.9
	Enc/Dec	**Prediction**	**Speedup**
Core exec. time	4.1ms + 3.3ms	9.5ms	155.6

Table 6. Performance measurement of ten liquid128 example scenes over 150 simulation steps each. The mean surface distance is a measure of deviation from the reference per solve[2].

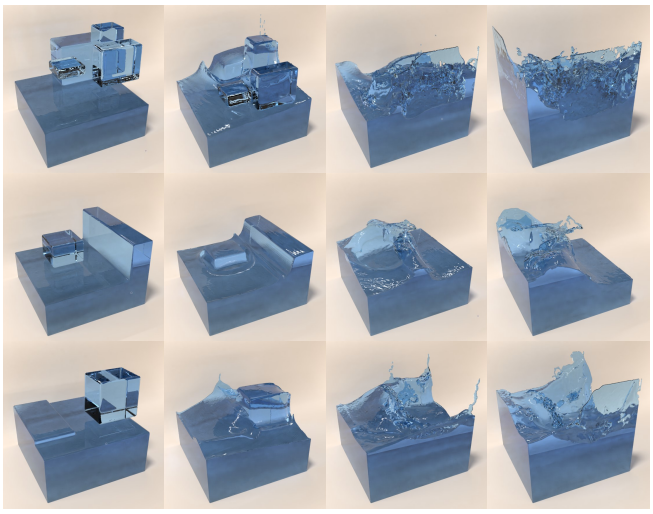


$t = 0$ $t = 50$ $t = 100$ $t = 150$

Fig. 11. Renderings at different points in time of a 64^3 scene predicted with $i_p = 4$ by our network.

7 DISCUSSION AND LIMITATIONS

While we have shown that our approach leads to large speed-ups and robust simulations for a significant variety of fluid scenes, it clearly has limitations, and there are numerous interesting extensions for future work. Among others, we are interested in exploring



$t = 0$ $t = 50$ $t = 100$ $t = 150$

Fig. 12. Several examples of 128^3 liquid scenes predicted with an interval of $i_p = 4$ by our LSTM network.

$t = 0$ $t = 33$ $t = 66$ $t = 99$

Fig. 13. Several examples of 128^3 smoke scenes predicted with an interval of $i_p = 3$ by our LSTM network.

larger spaces of flow behavior. E.g., interactions with obstacles, different time steps sizes, or different elasto-plastic materials would be interesting to incorporate.

In addition, improving the long-term prediction accuracy of our model is an important goal for future work. Our experiments show that larger data sets should directly translate into improved predictions. This is especially important for the latent space data set, which cannot be easily augmented. It would also be interesting to add additional constraints to the autoencoder to improve its reconstructions, e.g., for spatial smoothness, or divergence freeness. For the time predictions, it would be very interesting to investigate state-full LSTMs that preserve their state over the whole course of the prediction. In this way, our model could be run without resorting

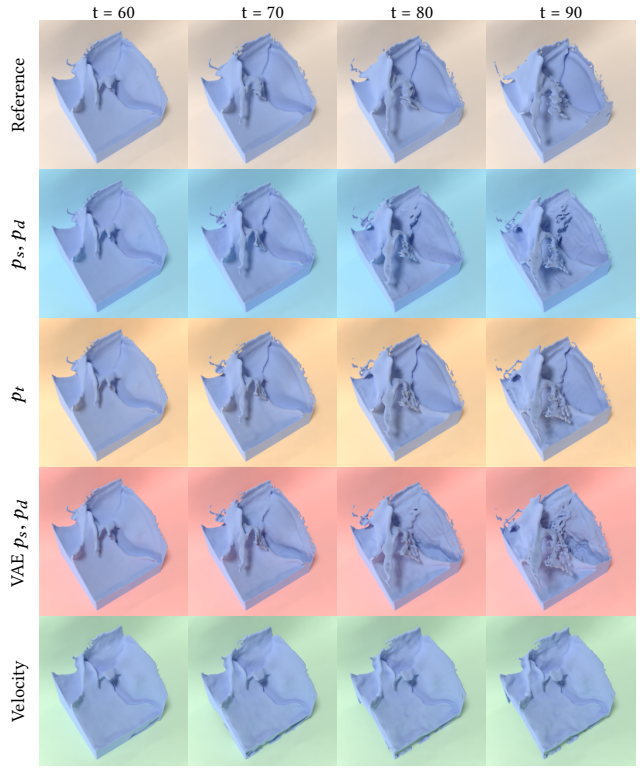


Fig. 14. Visual comparison of liquid surfaces predicted by different architectures for 40 time steps for $i_p = \infty$, with prediction starting at time step 50. While the velocity version (green) leads to large errors in surface position, all three pressure versions closely capture the large scale motions. On smaller scales, both split pressure and especially VAE introduce artifacts.

to a regular solver for preventing longer term drifts in the solution. In its current form, our model would however already work very well as a complement to CNN-based methods for divergence free projections [Tompson et al. 2016].

Finally, our temporal prediction model needs to be specifically trained for the physical effects in each data set. While the autoencoder can be used for differently sized inputs, directly using it for significantly larger inputs would lead to overly large latent-spaces. Instead, it would be preferable to add additional layers to the de- and encoder stacks, for which an existing, trained model is a good starting point.

8 CONCLUSION

We have explored a large variety of neural network architectures for space-time physics functions, and we demonstrated a first complete deep learning framework for temporal predictions of liquid pressure functions using latent space encodings. We realize our algorithm with a carefully constructed pair of networks: a spatial autoencoder and the temporal LSTM network. To prevent explosions in the number of weights for latent spaces with large dimensionality, we propose to use time-convolutions in conjunction with dense LSTM

layers. In this way, we arrive at a data-driven solver that yields practical speed-ups, and at its core is more than 150x faster than a regular pressure solve.

We believe that our work represents an important first step towards deep-learning powered simulation algorithms. We were able to show that it is feasible to infer dense, Eulerian functions with LSTM networks, and that despite all challenges, such algorithms can already yield practical gains in simulation performance. On the other hand, given the complexity of the problem at hand, our approach represents only a very first step. There are numerous, highly interesting avenues for future research, ranging from improving the accuracy of the predictions, over performance considerations, to using such physics predictions as priors for inverse problems, and we hope that our work will help to speed up advances in these directions.

REFERENCES

Ryoichi Ando, Nils Thuerey, and Chris Wojtan. 2015. A Dimension-reduced Pressure Solver for Liquid Simulations. *Comp. Grap. Forum* 34, 2 (2015), 10.

anonymous. 2018. Latent-space Physics: Towards Learning the Temporal Evolution of Fluid Flow. *non-peer-reviewed prepublication by the authors, arXiv:1801* (2018).

Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. 2016. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*. 4502–4510.

Christopher Batty, Florence Bertails, and Robert Bridson. 2007. A fast variational framework for accurate solid-fluid coupling. *ACM Trans. Graph.* 26, 3, Article 100 (July 2007). https://doi.org/10.1145/1276377.1276502

Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. 2007. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*. 153–160.

Hervé Bourlard and Yves Kamp. 1988. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics* 59, 4 (1988), 291–294.

Robert Bridson. 2015. *Fluid Simulation for Computer Graphics*. CRC Press.

Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. 2016. A compositional object-based approach to learning physical dynamics. *arXiv:1612.00341* (2016).

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Fethi Bougares, Holger Schwenk, and Y Bengio. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. (06 2014).

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

Sebastien Ehrhardt, Aron Monszpart, Niloy J Mitra, and Andrea Vedaldi. 2017. Learning A Physical Long-term Predictor. *arXiv:1703.00247* (2017).

Doug Enright, Duc Nguyen, Frederic Gibou, and Ron Fedkiw. 2003. Using the Particle Level Set Method and a Second Order Accurate Pressure Boundary Condition for Free-Surface Flows. *Proc. of the 4th ASME-JSME Joint Fluids Engineering Conference* (2003).

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

Yun Raymond Fei, Henrique Teles Maia, Christopher Batty, Changxi Zheng, and Eitan Grinspun. 2017. A multi-scale model for simulating liquid-hair interactions. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 56.

Florian Ferstl, Ryoichi Ando, Chris Wojtan, Rüdiger Westermann, and Nils Thuerey. 2016. Narrow band FLIP for liquid simulations. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 225–232.

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.

Nick Foster and Ronald Fedkiw. 2001. Practical animation of liquids. In *Proceedings of ACM SIGGRPAH*. 23–30.

Nick Foster and Dimitri Metaxas. 1996. Realistic Animation of Liquids. *Graphical Models and Image Processing* 58, 5 (Sept. 1996), 471–483. https://doi.org/10.1006/gmip.1996.0039

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

Alex Graves. 2013. Generating Sequences With Recurrent Neural Networks. *CoRR* abs/1308.0850 (2013). arXiv:1308.0850 http://arxiv.org/abs/1308.0850

Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *science* 313, 5786 (2006), 504–507.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780.

Daniel Holden, Taku Komura, and Jun Saito. 2017. Phase-functioned neural networks for character control. *ACM Trans. Graph.* 36, 4 (2017), 42.

Daniel P. Huttenlocher, Gregory A. Klanderman, and William J Rucklidge. 1993. Comparing images using the Hausdorff distance. *IEEE Transactions on pattern analysis and machine intelligence* 15, 9 (1993), 850–863.

Markus Ihmsen, Jens Cornelis, Barbara Solenthaler, Christopher Horvath, and Matthias Teschner. 2014. Implicit incompressible SPH. *IEEE Transactions on Visualization and Computer Graphics* 20, 3 (2014), 426–435.

M. Kass and G. Miller. 1990. Rapid, Stable Fluid Dynamics for Computer Graphics. *ACM Trans. Graph.* 24, 4 (1990), 49–55.

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.

Michael Lentine, Wen Zheng, and Ronald Fedkiw. 2010. A novel algorithm for incompressible flow using only a coarse grid projection. In *ACM Trans. Graph.*, Vol. 29. ACM, 114.

Yangyan Li, Angela Dai, Leonidas Guibas, and Matthias Nießner. 2015. Database-Assisted Object Retrieval for Real-Time 3D Reconstruction. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 435–446.

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).

Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. 2013. Rectifier Nonlinearities Improve Neural Network Acoustic Models.

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. 2014. Unified particle physics for real-time applications. *ACM Trans. Graph.* 33, 4 (2014), 153.

Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. 2011. Stacked convolutional auto-encoders for hierarchical feature extraction. *Artificial Neural Networks and Machine Learning–ICANN 2011* (2011), 52–59.

A. McAdams, E. Sifakis, and J. Teran. 2010. A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids. In *Symposium on Computer Animation (SCA '10)*. 65–74.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*. 1928–1937.

M. Müller, D. Charypar, and M. Gross. 2003. Particle-based Fluid Simulation for Interactive Applications. In *Symposium on Computer Animation*. 154–159.

Michael B. Nielsen, Brian B. Christensen, Nafees Bin Zafar, Doug Roble, and Ken Museth. 2009. Guiding of Smoke Animations through Variational Coupling of Simulations at Different Resolutions. In *Proceedings of the ACM SIGGRAPH/Eurographics symposium on Computer animation*. 217–226.

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).

Karthik Raveendran, Nils Thuerey, Chris Wojtan, and Greg Turk. 2014. Blending Liquids. *ACM Trans. Graph.* 33 (4) (August 2014), 10.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14)*. JMLR.org, II–1278–II–1286. http://dl.acm.org/citation.cfm?id=3044805.3045035

Avi Robinson-Mosher, Tamar Shinar, Jon Gretarsson, Jonathan Su, and Ronald Fedkiw. 2008. Two-way coupling of fluids to rigid and deformable solids and shells. In *ACM Transactions on Graphics (TOG)*, Vol. 27. ACM, 46.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1988. Learning representations by back-propagating errors. *Cognitive modeling* 5, 3 (1988), 1.

Shunsuke Saito, Lingyu Wei, Liwen Hu, Koki Nagano, and Hao Li. 2016. Photorealistic Facial Texture Inference Using Deep Neural Networks. *arXiv preprint arXiv:1612.00523* (2016).

Connor Schenck and Dieter Fox. 2017. Reasoning About Liquids via Closed-Loop Simulation. *arXiv:1703.01656* (2017).
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.
Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 3104–3112.
Yun Teng, David IW Levin, and Theodore Kim. 2016. Eulerian solid-fluid coupling. *ACM Trans. Graph.* 35, 6 (2016), 200.
Jonathan Tompson, Kristofer Schlachter, Pablo Sprechmann, and Ken Perlin. 2016. Accelerating Eulerian Fluid Simulation With Convolutional Networks. *arXiv: 1607.03597* (2016).
Adrien Treuille, Andrew Lewis, and Zoran Popović. 2006. Model reduction for real-time fluids. *ACM Trans. Graph.* 25, 3 (July 2006), 826–834.
Kiwon Um, Xiangyu Hu, and Nils Thuerey. 2017. Splash Modeling with Neural Networks. *arXiv:1704.04456* (2017).
Paul Upchurch, Jacob Gardner, Kavita Bala, Robert Pless, Noah Snavely, and Kilian Weinberger. 2016. Deep feature interpolation for image content changes. *arXiv preprint arXiv:1611.05507* (2016).
Nicholas Watters, Daniel Zoran, Theophane Weber, Peter Battaglia, Razvan Pascanu, and Andrea Tacchetti. 2017. Visual interaction networks. In *Advances in Neural Information Processing Systems*. 4540–4548.
Martin Wicke, Matthew Stanton, and Adrien Treuille. 2009. Modular Bases for Fluid Dynamics. *ACM Trans. Graph.* 28, 3 (Aug. 2009), 39.
Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. 2016. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *Advances in Neural Information Processing Systems*. 82–90.
Jun Xu, Ting Yao, Yongdong Zhang, and Tao Mei. 2017. Learning Multimodal Attention LSTM Networks for Video Captioning. In *Proceedings of the 2017 ACM on Multimedia Conference*. ACM, 537–545.
Cheng Yang, Xubo Yang, and Xiangyun Xiao. 2016. Data-driven projection method in fluid simulation. *Computer Animation and Virtual Worlds* 27, 3-4 (2016), 415–424.
Yongning Zhu and Robert Bridson. 2005. Animating Sand as a Fluid. *ACM Trans. Graph.* 24, 3 (2005), 965–972.

A LONG-SHORT TERM MEMORY UNITS

Below we briefly summarize the central equations for layers of LSTM units. Below, f, i, o, g, s will denote forget, input, output, update and result connections, respectively. h denotes the result of the LSTM layer. Subscripts denote time steps, while θ and b denote weight and bias. Below, we assume tanh as output activation function. The new state for time step t of an LSTM layer is then given by:

$$\begin{aligned} f_t &= \sigma(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f) \\ i_t &= \sigma(\theta_{xi} x_t + \theta_{hi} h_{t-1} + b_i) \\ o_t &= \sigma(\theta_{xo} x_t + \theta_{ho} h_{t-1} + b_o) \\ g_t &= \tanh(\theta_{xg} x_t + \theta_{hg} h_{t-1} + b_g) \\ s_t &= f_t \odot s_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(s_t) \end{aligned} \tag{8}$$

Correspondingly, the number of weights of such a layer with n_o nodes, i.e., outputs, and n_i inputs is given by $n_{\text{lstm}} = 4(n_o^2 + n_o(n_i + 1))$.

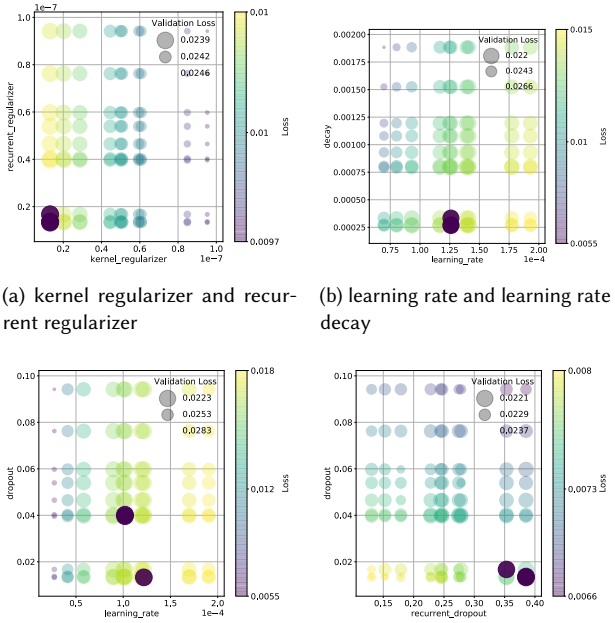
In contrast, the number of weights for the 1D convolutions we propose is $n_{\text{conv-1d}} = n_o k(n_i + 1)$, with a kernel size $k = 1$.

B HYPERPARAMETERS

To find appropriate hyperparameters for the prediction network, a large number of training runs with varying parameters were executed on a subset of the total training data domain. The subset consisted of 100 scenes of the training data set discussed in Sec. 4.3.

In Fig. 15 (a-d), examples for those searches are shown. Each circle shown in the graphs represents the final result of one complete training run with the parameters given on the axes. The color represents the mean absolute error of the training error, ranging from purple (the best) to yellow (the worst). The size of the circle corresponds to the validation error, i.e., the most important quantity we are interested in. The best two runs are highlighted with a dark coloring. These searches yield interesting results, e.g. Fig. 15a shows that the network performed best without any weight decay regularization applied.

Choosing good parameters leads to robust learning behavior in the training process, an example is shown in Fig. 16. Note that it is possible for the validation error to be lower than the training error as dropout is not used for computing the validation loss. The mean absolute error of the prediction on a test set of 40 scenes, which was generated independently from the training and validation data, was 0.0201 for this case. These results suggest that the network generalizes well with the given hyperparameters.



(a) kernel regularizer and recurrent regularizer

(b) learning rate and learning rate decay

(c) learning rate and dropout

(d) recurrent dropout and dropout

Fig. 15. Random search of hyperparameters for the prediction network

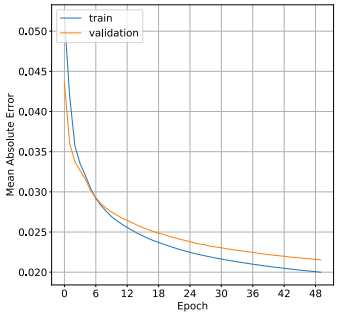


Fig. 16. Training history of the liquid128 total pressure prediction network.