Learning to Simulate Complex Physics with Graph Networks

Alvaro Sanchez-Gonzalez [*1] **Jonathan Godwin** [*1] **Tobias Pfaff** [*1] **Rex Ying** [*1 2] **Jure Leskovec** [2]
Peter W. Battaglia [1]

Abstract

Here we present a machine learning framework and model implementation that can learn to simulate a wide variety of challenging physical domains, involving fluids, rigid solids, and deformable materials interacting with one another. Our framework—which we term "Graph Network-based Simulators" (GNS)—represents the state of a physical system with particles, expressed as nodes in a graph, and computes dynamics via learned message-passing. Our results show that our model can generalize from single-timestep predictions with thousands of particles during training, to different initial conditions, thousands of timesteps, and at least an order of magnitude more particles at test time. Our model was robust to hyperparameter choices across various evaluation metrics: the main determinants of long-term performance were the number of message-passing steps, and mitigating the accumulation of error by corrupting the training data with noise. Our GNS framework advances the state-of-the-art in learned physical simulation, and holds promise for solving a wide range of complex forward and inverse problems.

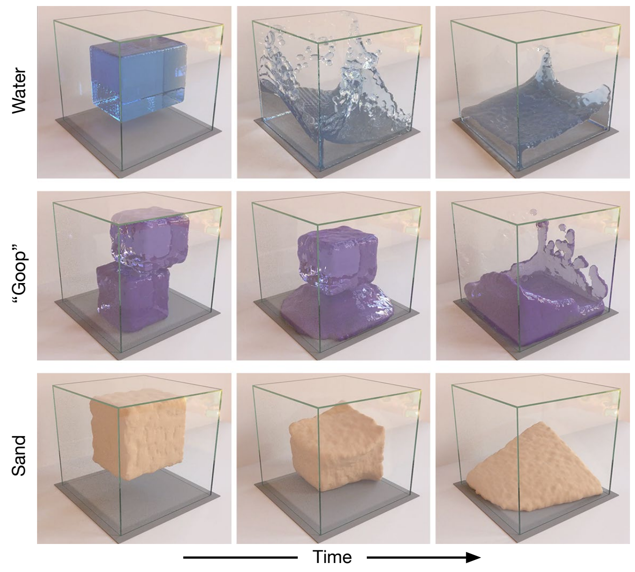

Figure 1. Rollouts of our GNS model for our WATER-3D, GOOP-3D and SAND-3D datasets. It learns to simulate rich materials at resolutions sufficient for high-quality rendering [video].

1. Introduction

Realistic simulators of complex physics are invaluable to many scientific and engineering disciplines, however traditional simulators can be very expensive to create and use. Building a simulator can entail years of engineering effort, and often must trade off generality for accuracy in a narrow range of settings. High-quality simulators require substantial computational resources, which makes scaling up prohibitive. Even the best are often inaccurate due to insufficient knowledge of, or difficulty in approximating, the underlying physics and parameters. An attractive alternative to traditional simulators is to use machine learning to train simulators directly from observed data, however the large state spaces and complex dynamics have been difficult for standard end-to-end learning approaches to overcome.

Here we present a powerful machine learning framework for learning to simulate complex systems from data—"Graph Network-based Simulators" (GNS). Our framework imposes strong inductive biases, where rich physical states are represented by graphs of interacting particles, and complex dynamics are approximated by learned message-passing among nodes.

We implemented our GNS framework in a single deep learning architecture, and found it could learn to accurately simulate a wide range of physical systems in which fluids, rigid solids, and deformable materials interact with one another. Our model also generalized well to much larger systems and

[*]Equal contribution [1]DeepMind, London, UK [2]Department of Computer Science, Stanford University, Stanford, CA, USA. Email to: alvarosg@google.com, jonathangodwin@google.com, tpfaff@google.com, rexying@stanford.edu, jure@cs.stanford.edu, peterbattaglia@google.com.

Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

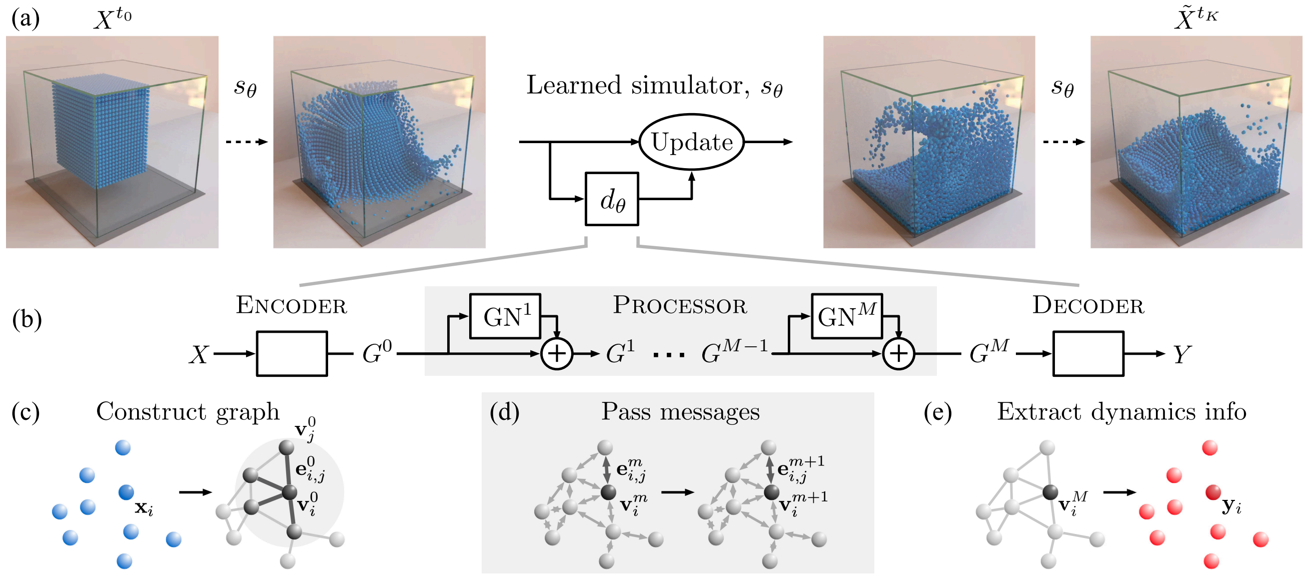

Figure 2. **(a)** Our GNS predicts future states represented as particles using its learned dynamics model, d_θ, and a fixed update procedure. **(b)** The d_θ uses an "encode-process-decode" scheme, which computes dynamics information, Y, from input state, X. **(c)** The ENCODER constructs latent graph, G^0, from the input state, X. **(d)** The PROCESSOR performs M rounds of learned message-passing over the latent graphs, $G^0, \ldots, G^M$. **(e)** The DECODER extracts dynamics information, Y, from the final latent graph, G^M.

longer time scales than those on which it was trained. While previous learning simulation approaches (Li et al., 2018; Ummenhofer et al., 2020) have been highly specialized for particular tasks, we found our single GNS model performed well across dozens of experiments and was generally robust to hyperparameter choices. Our analyses showed that performance was determined by a handful of key factors: its ability to compute long-range interactions, inductive biases for spatial invariance, and training procedures which mitigate the accumulation of error over long simulated trajectories.

2. Related Work

Our approach focuses on *particle-based* simulation, which is used widely across science and engineering, e.g., computational fluid dynamics, computer graphics. States are represented as a set of particles, which encode mass, material, movement, etc. within local regions of space. Dynamics are computed on the basis of particles' interactions within their local neighborhoods. One popular particle-based method for simulating fluids is "smoothed particle hydrodynamics" (SPH) (Monaghan, 1992), which evaluates pressure and viscosity forces around each particle, and updates particles' velocities and positions accordingly. Other techniques, such as "position-based dynamics" (PBD) (Müller et al., 2007) and "material point method" (MPM) (Sulsky et al., 1995), are more suitable for interacting, deformable materials. In PBD, incompressibility and collision dynamics involve resolving pairwise distance constraints between particles, and directly predicting their position changes. Several differentiable particle-based simulators have recently appeared, e.g., DiffTaichi (Hu et al., 2019), PhiFlow (Holl et al., 2020), and Jax-MD (Schoenholz & Cubuk, 2019), which can backpropagate gradients through the architecture.

Learning simulations from data (Grzeszczuk et al., 1998) has been an important area of study with applications in physics and graphics. Compared to engineered simulators, a learned simulator can be far more efficient for predicting complex phenomena (He et al., 2019); e.g., (Ladický et al., 2015; Wiewel et al., 2019) learn parts of a fluid simulator for faster prediction.

Graph Networks (GN) (Battaglia et al., 2018)—a type of graph neural network (Scarselli et al., 2008)—have recently proven effective at learning forward dynamics in various settings that involve interactions between many entities. A GN maps an input graph to an output graph with the same structure but potentially different node, edge, and graph-level attributes, and can be trained to learn a form of message-passing (Gilmer et al., 2017), where latent information is propagated between nodes via the edges. GNs and their variants, e.g., "interaction networks", can learn to simulate rigid body, mass-spring, n-body, and robotic control systems (Battaglia et al., 2016; Chang et al., 2016; Sanchez-Gonzalez et al., 2018; Mrowca et al., 2018; Li et al., 2019; Sanchez-Gonzalez et al., 2019), as well as non-physical systems, such as multi-agent dynamics (Tacchetti et al., 2018; Sun et al., 2019), algorithm execution (Veličković et al., 2020), and other dynamic graph settings (Trivedi et al., 2019; 2017; Yan et al., 2018; Manessi et al., 2020).

Our GNS framework builds on and generalizes several lines of work, especially Sanchez-Gonzalez et al. (2018)'s GN-based model which was applied to various robotic control systems, Li et al. (2018)'s DPI which was applied to fluid dynamics, and Ummenhofer et al. (2020)'s Continuous Convolution (CConv) which was presented as a non-graph-based method for simulating fluids. Crucially, our GNS framework is a *general* approach to learning simulation, is simpler to implement, and is more accurate across fluid, rigid, and deformable material systems.

3. GNS Model Framework

3.1. General Learnable Simulation

We assume $X^t \in \mathcal{X}$ is the state of the world at time t. Applying physical dynamics over K timesteps yields a trajectory of states, $\mathbf{X}^{t_{0:K}} = (X^{t_0}, \dots, X^{t_K})$. A *simulator*, $s : \mathcal{X} \to \mathcal{X}$, models the dynamics by mapping preceding states to causally consequent future states. We denote a simulated "rollout" trajectory as, $\tilde{\mathbf{X}}^{t_{0:K}} = (X^{t_0}, \tilde{X}^{t_1}, \dots, \tilde{X}^{t_K})$, which is computed iteratively by, $\tilde{X}^{t_{k+1}} = s(\tilde{X}^{t_k})$ for each timestep. Simulators compute dynamics information that reflects how the current state is changing, and use it to update the current state to a predicted future state (see Figure 2(a)). An example is a numerical differential equation solver: the equations compute dynamics information, i.e., time derivatives, and the integrator is the update mechanism.

A learnable simulator, s_θ, computes the dynamics information with a parameterized function approximator, $d_\theta : \mathcal{X} \to \mathcal{Y}$, whose parameters, θ, can be optimized for some training objective. The $Y \in \mathcal{Y}$ represents the dynamics information, whose semantics are determined by the update mechanism. The update mechanism can be seen as a function which takes the $\tilde{X}^{t_k}$, and uses d_θ to predict the next state, $\tilde{X}^{t_{k+1}} = \text{Update}(\tilde{X}^{t_k}, d_\theta)$. Here we assume a simple update mechanism—an Euler integrator—and $\mathcal{Y}$ that represents accelerations. However, more sophisticated update procedures which call d_θ more than once can also be used, such as higher-order integrators (e.g., Sanchez-Gonzalez et al. (2019)).

3.2. Simulation as Message-Passing on a Graph

Our learnable simulation approach adopts a particle-based representation of the physical system (see Section 2), i.e., $X = (\mathbf{x}_0, \dots, \mathbf{x}_N)$, where each of the N particles' $\mathbf{x}_i$ represents its state. Physical dynamics are approximated by interactions among the particles, e.g., exchanging energy and momentum among their neighbors. The way particle-particle interactions are modeled determines the quality and generality of a simulation method—i.e., the types of effects and materials it can simulate, in which scenarios the method performs well or poorly, etc. We are interested in learning these interactions, which should, in principle, allow learning the dynamics of any system that can be expressed as particle dynamics. So it is crucial that different θ values allow d_θ to span a wide range of particle-particle interaction functions.

Particle-based simulation can be viewed as message-passing on a graph. The nodes correspond to particles, and the edges correspond to pairwise relations among particles, over which interactions are computed. We can understand methods like SPH in this framework—the messages passed between nodes could correspond to, e.g., evaluating pressure using the density kernel.

We capitalize on the correspondence between particle-based simulators and message-passing on graphs to define a general-purpose d_θ based on GNs. Our d_θ has three steps—ENCODER, PROCESSOR, DECODER (Battaglia et al., 2018) (see Figure 2(b)).

ENCODER definition. The $\text{ENCODER} : \mathcal{X} \to \mathcal{G}$ embeds the particle-based state representation, X, as a latent graph, $G^0 = \text{ENCODER}(X)$, where $G = (V, E, \mathbf{u})$, $\mathbf{v}_i \in V$, and $\mathbf{e}_{i,j} \in E$ (see Figure 2(b,c)). The node embeddings, $\mathbf{v}_i = \varepsilon^v(\mathbf{x}_i)$, are learned functions of the particles' states. Directed edges are added to create paths between particle nodes which have some potential interaction. The edge embeddings, $\mathbf{e}_{i,j} = \varepsilon^e(\mathbf{r}_{i,j})$, are learned functions of the pairwise properties of the corresponding particles, $\mathbf{r}_{i,j}$, e.g., displacement between their positions, spring constant, etc. The graph-level embedding, $\mathbf{u}$, can represent global properties such as gravity and magnetic fields (though in our implementation we simply appended those as input node features—see Section 4.2 below).

PROCESSOR definition. The $\text{PROCESSOR} : \mathcal{G} \to \mathcal{G}$ computes interactions among nodes via M steps of learned message-passing, to generate a sequence of updated latent graphs, $\mathbf{G} = (G^1, ..., G^M)$, where $G^{m+1} = \text{GN}^{m+1}(G^m)$ (see Figure 2(b,d)). It returns the final graph, $G^M = \text{PROCESSOR}(G^0)$. Message-passing allows information to propagate and constraints to be respected: the number of message-passing steps required will likely scale with the complexity of the interactions.

DECODER definition. The $\text{DECODER} : \mathcal{G} \to \mathcal{Y}$ extracts dynamics information from the nodes of the final latent graph, $\mathbf{y}_i = \delta^v(\mathbf{v}_i^M)$ (see Figure 2(b,e)). Learning δ^v should cause the $\mathcal{Y}$ representations to reflect relevant dynamics information, such as acceleration, in order to be semantically meaningful to the update procedure.

4. Experimental Methods

Code and data available at github.com/deepmind/deepmind-research/tree/master/learning_to_simulate.

4.1. Physical Domains

We explored how our GNS learns to simulate in datasets which contained three diverse, complex physical materials: water as a barely damped fluid, chaotic in nature; sand as a granular material with complex frictional behavior; and "goop" as a viscous, plastically deformable material. These materials have very different behavior, and in most simulators, require implementing separate material models or even entirely different simulation algorithms.

For one domain, we use Li et al. (2018)'s BOXBATH, which simulates a container of water and a cube floating inside, all represented as particles, using the PBD engine FleX (Macklin et al., 2014).

We also created WATER-3D, a high-resolution 3D water scenario with randomized water position, initial velocity and volume, comparable to Ummenhofer et al. (2020)'s containers of water. We used SPlisHSPlasH (Bender & Koschier, 2015), a SPH-based fluid simulator with strict volume preservation to generate this dataset.

For most of our domains, we use the Taichi-MPM engine (Hu et al., 2018) to simulate a variety of challenging 2D and 3D scenarios. We chose MPM for the simulator because it can simulate a very wide range of materials, and also has some different properties than PBD and SPH, e.g., particles may become compressed over time.

Our datasets typically contained 1000 train, 100 validation and 100 test trajectories, each simulated for 300-2000 timesteps (tailored to the average duration for the various materials to come to a stable equilibrium). A detailed listing of all our datasets can be found in the Supplementary Materials B.

4.2. GNS Implementation Details

We implemented the components of the GNS framework using standard deep learning building blocks, and used standard nearest neighbor algorithms (Dong et al., 2011; Chen et al., 2009; Tang et al., 2016) to construct the graph.

Input and output representations. Each particle's input state vector represents position, a sequence of $C = 5$ previous velocities[1], and features that capture static material properties (e.g., water, sand, goop, rigid, boundary particle), $\mathbf{x}_i^{t_k} = [\mathbf{p}_i^{t_k}, \dot{\mathbf{p}}_i^{t_{k-C+1}}, \dots, \dot{\mathbf{p}}_i^{t_k}, \mathbf{f}_i]$, respectively. The global properties of the system, $\mathbf{g}$, include external forces and global material properties, when applicable. The prediction targets for supervised learning are the per-particle average acceleration, $\ddot{\mathbf{p}}_i$. Note that in our datasets, we only require $\mathbf{p}_i$ vectors: the $\dot{\mathbf{p}}_i$ and $\ddot{\mathbf{p}}_i$ are computed from $\mathbf{p}_i$ using finite differences. For full details of these input and target features, see Supplementary Material Section B.

ENCODER details. The ENCODER constructs the graph structure G^0 by assigning a node to each particle and adding edges between particles within a "connectivity radius", R, which reflected local interactions of particles, and which was kept constant for all simulations of the same resolution. For generating rollouts, on each timestep the graph's edges were recomputed by a nearest neighbor algorithm, to reflect the current particle positions.

The ENCODER implements ε^v and ε^e as multilayer perceptrons (MLP), which encode node features and edge features into the latent vectors, $\mathbf{v}_i$ and $\mathbf{e}_{i,j}$, of size 128.

We tested two ENCODER variants, distinguished by whether they use absolute versus relative positional information. For the absolute variant, the input to ε^v was the $\mathbf{x}_i$ described above, with the globals features concatenated to it. The input to ε^e, i.e., $\mathbf{r}_{i,j}$, did not actually carry any information and was discarded, with the $\mathbf{e}_i^0$ in G^0 set to a trainable fixed bias vector. The relative ENCODER variant was designed to impose an inductive bias of invariance to absolute spatial location. The ε^v was forced to ignore $\mathbf{p}_i$ information within $\mathbf{x}_i$ by masking it out. The ε^e was provided with the relative positional displacement, and its magnitude[2], $\mathbf{r}_{i,j} = [(\mathbf{p}_i - \mathbf{p}_j), \|\mathbf{p}_i - \mathbf{p}_j\|]$. Both variants concatenated the global properties $\mathbf{g}$ onto each $\mathbf{x}_i$ before passing it to ε^v.

PROCESSOR details. Our processor uses a stack of M GNs (where M is a hyperparameter) with identical structure, MLPs as internal edge and node update functions, and either shared or unshared parameters (as analyzed in Results Section 5.4). We use GNs without global features or global updates (similar to an interaction network)[3], and with a residual connections between the input and output latent node and edge attributes.

DECODER details. Our decoder's learned function, δ^v, is an MLP. After the DECODER, the future position and velocity are updated using an Euler integrator, so the $\mathbf{y}_i$ corresponds to accelerations, $\ddot{\mathbf{p}}_i$, with 2D or 3D dimension, depending on the physical domain. As mentioned above, the supervised training outputs were simply these, $\ddot{\mathbf{p}}_i$ vectors[4].

Neural network parameterizations. All MLPs have two hidden layers (with ReLU activations), followed by a non-

[1] C is a hyperparameter which we explore in our experiments.

[2] Similarly, relative velocities could be used to enforce invariance to inertial frames of reference.

[3] In preliminary experiments we also attempted using a PROCESSOR with a full GN and a global latent state, for which the global features $\mathbf{g}$ are encoded with a separate ε^g MLP.

[4] Note that in this case optimizing for acceleration is equivalent to optimizing for position, because the acceleration is computed as first order finite difference from the position and we use an Euler integrator to update the position.

activated output layer, each layer with size of 128. All MLPs (except the output decoder) are followed by a LayerNorm (Ba et al., 2016) layer, which we generally found improved training stability.

4.3. Training

Software. We implemented our models using TensorFlow 1, Sonnet 1, and the "Graph Nets" library (2018).

Training noise. Modeling a complex and chaotic simulation system requires the model to mitigate error accumulation over long rollouts. Because we train our models on ground-truth one-step data, they are never presented with input data corrupted by this sort of accumulated noise. This means that when we generate a rollout by feeding the model with its own noisy, previous predictions as input, the fact that its inputs are outside the training distribution may lead it to make more substantial errors, and thus rapidly accumulate further error. We use a simple approach to make the model more robust to noisy inputs: at training we corrupt the input velocities of the model with random-walk noise $\mathcal{N}(0, \sigma_v = 0.0003)$ (adjusting input positions), so the training distribution is closer to the distribution generated during rollouts. See Supplementary Materials B for full details.

Normalization. We normalize all input and target vectors elementwise to zero mean and unit variance, using statistics computed online during training. Preliminary experiments showed that normalization led to faster training, though converged performance was not noticeably improved.

Loss function and optimization procedures. We randomly sampled particle state pairs $(\mathbf{x}_i^{t_k}, \mathbf{x}_i^{t_{k+1}})$ from training trajectories, calculated target accelerations $\ddot{\mathbf{p}}_i^{t_k}$ (subtracting the noise added to the most recent input velocity), and computed the L_2 loss on the predicted per-particle accelerations, i.e., $L(\mathbf{x}_i^{t_k}, \mathbf{x}_i^{t_{k+1}}; \theta) = \|d_\theta(\mathbf{x}_i^{t_k}) - \ddot{\mathbf{p}}_i^{t_k}\|^2$. We optimized the model parameters θ over this loss with the Adam optimizer (Kingma & Ba, 2014), using a nominal[5] mini-batch size of 2. We performed a maximum of 20M gradient update steps, with exponential learning rate decay from 10^{-4} to 10^{-6}. While models can train in significantly less steps, we avoid aggressive learning rates to reduce variance across datasets and make comparisons across settings more fair.

We evaluated our models regularly during training by producing full-length rollouts on 5 held-out validation trajectories, and recorded the associated model parameters for best rollout MSE. We stopped training when we observed negligible decrease in MSE, which, on GPU/TPU hardware, was typically within a few hours for smaller, simpler datasets, and up to a week for the larger, more complex datasets.

[5]The actual batch size varies at each step dynamically. See Supplementary Material for more details.

Experimental domain	N	K	**1-step** ($\times 10^{-9}$)	**Rollout** ($\times 10^{-3}$)
WATER-3D (SPH)	13k	800	8.66	10.1
SAND-3D	20k	350	1.42	0.554
GOOP-3D	14k	300	1.32	0.618
WATER-3D-S (SPH)	5.8k	800	9.66	9.52
BOXBATH (PBD)	1k	150	54.5	4.2
WATER	1.9k	1000	2.82	17.4
SAND	2k	320	6.23	2.37
GOOP	1.9k	400	2.91	1.89
MULTIMATERIAL	2k	1000	1.81	16.9
FLUIDSHAKE	1.3k	2000	2.1	20.1
WATERDROP	1k	1000	1.52	7.01
WATERDROP-XL	7.1k	1000	1.23	14.9
WATERRAMPS	2.3k	600	4.91	11.6
SANDRAMPS	3.3k	400	2.77	2.07
RANDOMFLOOR	3.4k	600	2.77	6.72
CONTINUOUS	4.3k	400	2.06	1.06

Table 1. List of maximum number of particles N, sequence length K, and quantitative model accuracy (MSE) on the held-out test set. All domain names are also hyperlinks to the video website.

4.4. Evaluation

To report quantitative results, we evaluated our models after training converged by computing one-step and rollout metrics on held-out test trajectories, drawn from the same distribution of initial conditions used for training. We used particle-wise MSE as our main metric between ground truth and predicted data, both for rollout and one-step predictions, averaging across time, particle and spatial axes. We also investigated distributional metrics including optimal transport (OT) (Villani, 2003) (approximated by the Sinkhorn Algorithm (Cuturi, 2013)), and Maximum Mean Discrepancy (MMD) (Gretton et al., 2012). For the generalization experiments we also evaluate our models on a number of initial conditions drawn from distributions different than those seen during training, including, different number of particles, different object shapes, different number of objects, different initial positions and velocities and longer trajectories. See Supplementary Materials B for full details on metrics and evaluation.

5. Results

Our main findings are that our GNS model can learn accurate, high-resolution, long-term simulations of different fluids, deformables, and rigid solids, and it can generalize well beyond training to much longer, larger, and challenging settings. In Section 5.5 below, we compare our GNS model to two recent, related approaches, and find our approach was simpler, more generally applicable, and more accurate.

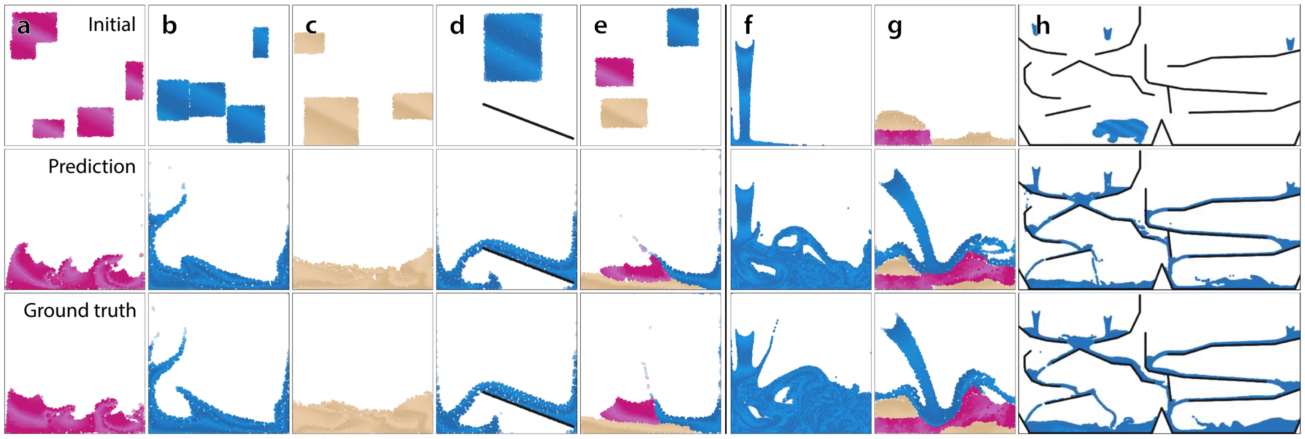

Figure 3. We can simulate many materials, from **(a)** GOOP over **(b)** WATER to **(c)** SAND, and **(d)** their interaction with rigid obstacles (WATERRAMPS). We can even train a single model on **(e)** multiple materials and their interaction (MULTIMATERIAL). We applied pre-trained models on several out-of-distribution tasks, involving **(f)** high-res turbulence (trained on WATERRAMPS), **(g)** multi-material interactions with unseen objects (trained on MULTIMATERIAL), and **(h)** generalizing on significantly larger domains (trained on WATERRAMPS). In the two bottom rows, we show a comparison of our model's prediction with the ground truth on the final frame for goop and sand, and on a representative mid-trajectory frame for water.

To challenge the robustness of our architecture, we used a single set of model hyperparameters for training across all of our experiments. Our GNS architecture used the relative ENCODER variant, 10 steps of message-passing, with unshared GN parameters in the PROCESSOR. We applied noise with a scale of $3 \cdot 10^{-4}$ to the input states during training.

5.1. Simulating Complex Materials

Our GNS model was very effective at learning to simulate different complex materials. Table 1 shows the one-step and rollout accuracy, as MSE, for all experiments. For intuition about what these numbers mean, the edge length of the container was approximately 1.0, and Figure 3(a-c) shows rendered images of the rollouts of our model, compared to ground truth[6]. Visually, the model's rollouts are quite plausible. Though specific model-generated trajectories can be distinguished from ground truth when compared side-by-side, it is difficult to visually classify individual videos as generated from our model versus the ground truth simulator.

Our GNS model scales to large amounts of particles and very long rollouts. With up to 19k particles in our 3D domains—substantially greater than demonstrated in previous methods—GNS can operate at resolutions high enough for practical prediction tasks and high-quality 3D renderings (e.g., Figure 1). And although our models were trained to make one-step predictions, the long-term trajectories remain plausible even over thousands of rollout timesteps.

[6]All rollout videos can be found here: `https://sites.google.com/view/learning-to-simulate`

The GNS model could also learn how the materials respond to unpredictable external forces. In the FLUIDSHAKE domain, a container filled with water is being moved side-to-side, causing splashes and irregular waves.

Our model could also simulate fluid interacting with complicated static obstacles, as demonstrated by our WATERRAMPS and SANDRAMPS domains in which water or sand pour over 1-5 obstacles. Figure 3(d) depicts comparisons between our model and ground truth, and Table 1 shows quantitative performance measures.

We also trained our model on continuously varying material parameters. In the CONTINUOUS domain, we varied the friction angle of a granular material, to yield behavior similar to a liquid ($0°$), sand ($45°$), or gravel ($> 60°$). Our results and videos show that our model can account for these continuous variations, and even interpolate between them: a model trained with the region $[30°, 55°]$ held out in training can accurately predict within that range. Additional quantitative results are available in Supplementary Materials C.

5.2. Multiple Interacting Materials

So far we have reported results of training identical GNS architectures separately on different systems and materials. However, we found we could go a step further and train a single architecture with a single set of parameters to simulate all of our different materials, interacting with each other in a single system.

In our MULTIMATERIAL domain, the different materials could interact with each other in complex ways, which means the model had to effectively learn the product space of different interactions (e.g., water-water, sand-sand, water-sand, etc.). The behavior of these systems was often much richer than the single-material domains: the stiffer materials, such as sand and goop, could form temporary semi-rigid obstacles, which the water would then flow around. Figure 3(e) and this video shows renderings of such rollouts. Visually, our model's performance in MULTIMATERIAL is comparable to its performance when trained on those materials individually.

5.3. Generalization

We found that the GNS generalizes well even beyond its training distributions, which suggests it learns a more general-purpose understanding of the materials and physical processes experienced during training.

To examine its capacity for generalization, we trained a GNS architecture on WATERRAMPS, whose initial conditions involved a square region of water in a container, with 1-5 ramps of random orientation and location. After training, we tested the model on several very different settings. In one generalization condition, rather than all water being present in the initial timestep, we created an "inflow" that continuously added water particles to the scene during the rollout, as shown in Figure 3(f). When unrolled for 2500 time steps, the scene contained 28k particles—an order of magnitude more than the 2.5k particles used in training—and the model was able to predict complex, highly chaotic dynamics not experienced during training, as can be seen in this video. The predicted dynamics were visually similar to the ground truth sequence.

Because we used relative displacements between particles as input to our model, in principle the model should handle scenes with much larger spatial extent at test time. We evaluated this on a much larger domain, with several inflows over a complicated arrangement of slides and ramps (see Figure 3(h), video here). The test domain's spatial width $\times$ height were 8.0×4.0, which was 32x larger than the training domain's area; at the end of the rollout, the number of particles was 85k, which was 34x more than during training; we unrolled the model for 5000 steps, which was 8x longer than the training trajectories. We conducted a similar experiment with sand on the SANDRAMPS domain, testing model generalization to hourglass-shaped ramps.

As a final, extreme test of generalization, we applied a model trained on MULTIMATERIAL to a custom test domain with inflows of various materials and shapes (Figure 3(g)). The model learned about frictional behavior between different materials (sand on sticky goop, versus slippery floor), and that the model generalized well to unseen shapes, such as hippo-shaped chunks of goop and water, falling from mid-air, as can be observed in this video.

5.4. Key Architectural Choices

We performed a comprehensive analysis of our GNS's architectural choices to discover what influenced performance most heavily. We analyzed a number of hyperparameter choices—e.g., number of MLP layers, linear encoder and decoder functions, global latent state in the PROCESSOR—but found these had minimal impact on performance (see Supplementary Materials C for details).

While our GNS model was generally robust to architectural and hyperparameter settings, we also identified several factors which had more substantial impact:

1. the number of message-passing steps,
2. shared vs. unshared PROCESSOR GN parameters,
3. the connectivity radius,
4. the scale of noise added to the inputs during training,
5. relative vs. absolute ENCODER.

We varied these choices systematically for each axis, fixing all other axes with the default architecture's choices, and report their impact on model performance in the GOOP domain (Figure 4).

For (1), Figure 4(a,b) shows that a greater number of message-passing steps M yielded improved performance in both one-step and rollout accuracy. This is likely because increasing M allows computing longer-range, and more complex, interactions among particles. Because computation time scales linearly with M, in practice it is advisable to use the smallest M that still provides desired performance.

For (2), Figure 4(c,d) shows that models with unshared GN parameters in the PROCESSOR yield better accuracy, especially for rollouts. Shared parameters imposes a strong inductive bias that makes the PROCESSOR analogous to a recurrent model, while unshared parameters are more analogous to a deep architecture, which incurs M times more parameters. In practice, we found marginal difference in computational costs or overfitting, so we conclude that using unshared parameters has little downside.

For (3), Figure 4(e,f) shows that greater connectivity R values yield lower error. Similar to increasing M, larger neighborhoods allow longer-range communication among nodes. Since the number of edges increases with R, more computation and memory is required, so in practice the minimal R that gives desired performance should be used.

For (4), we observed that rollout accuracy is best for an intermediate noise scale (see Figure 4(g,h)), consistent with our motivation for using it (see Section 4.3). We also note that one-step accuracy decreases with increasing noise scale. This is not surprising: adding noise makes the training distribution less similar to the uncorrupted distribution used for one-step evaluation.

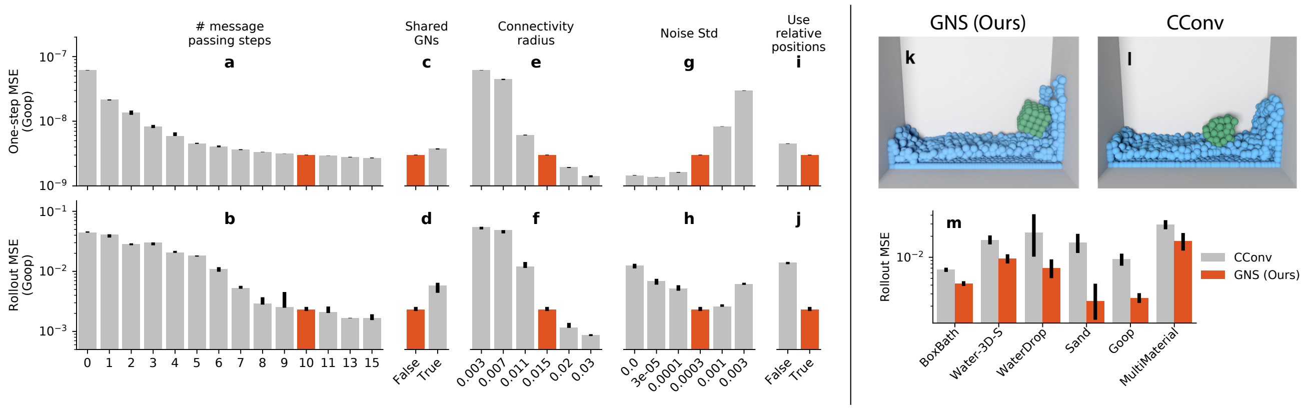

Figure 4. (left) Effect of different ablations (grey) against our model (red) on the one-step error **(a,c,e,g,i)** and the rollout error **(b,d,f,h,j)**. Bars show the median seed performance averaged across the entire GOOP test dataset. Error bars display lower and higher quartiles, and are shown for the default parameters. (right) Comparison of average performance of our GNS model to CConv. **(k,l)** Qualitative comparison between GNS **(k)** and CConv **(l)** in BOXBATH after 50 rollout steps (video link). **(m)** Quantitative comparison of our GNS model (red) to the CConv model (grey) across the test set . For our model, we trained one or more seeds using the same set of hyper-parameters and show results for all seeds. For the CConv model we ran several variations including different radius sizes, noise levels, and number of unroll steps during training, and show the result for the best seed. Errors bars show the standard error of the mean across all of the trajectories in the test set (95% confidence level).

For (5), Figure 4(i,j) shows that the relative ENCODER is clearly better than the absolute version. This is likely because the underlying physical processes that are being learned are invariant to spatial position, and the relative ENCODER's inductive bias is consistent with this invariance.

5.5. Comparisons to Previous Models

We compared our approach to two recent papers which explored learned fluid simulators using particle-based approaches. Li et al. (2018)'s DPI studied four datasets of fluid, deformable, and solid simulations, and presented four different, distinct architectures, which were similar to Sanchez-Gonzalez et al. (2018)'s, with additional features such as hierarchical latent nodes. When training our GNS model on DPI's BOXBATH domain, we found it could learn to simulate the rigid solid box floating in water, faithfully maintaining the stiff relative displacements among rigid particles, as shown Figure 4(k) and this video. Our GNS model did not require any modification—the box particles' material type was simply a feature in the input vector—while DPI required a specialized hierarchical mechanism and forced all box particles to preserve their relative displacements with each other. Presumably the relative ENCODER and training noise alleviated the need for such mechanisms.

Ummenhofer et al. (2020)'s CConv propagates information across particles[7], and uses particle update functions and training procedures which are carefully tailored to modeling fluid dynamics (e.g., an SPH-like local kernel, different sub-networks for fluid and boundary particles, a loss function that weights slow particles with few neighbors more heavily). Ummenhofer et al. (2020) reported CConv outperformed DPI, so we quantitatively compared our GNS model to CConv. We implemented CConv as described in its paper, plus two additional versions which borrowed our noise and multiple input states, and performed hyperparameter sweeps over various CConv parameters. Figure 4(m) shows that across all six domains we tested, our GNS model with default hyperparameters has better rollout accuracy than the best CConv model (among the different versions and hyperparameters) for that domain. In this comparison video, we observe than CConv performs well for domains like water, which it was built for, but struggles with some of our more complex materials. Similarly, in a CConv rollout of the BOXBATH DOMAIN the rigid box loses its shape (Figure 4(l)), while our method preserves it. See Supplementary Materials D for full details of our DPI and CConv comparisons.

[7]The authors state CConv does not use an explicit graph representation, however we believe their particle update scheme can be interpreted as a special type of message-passing on a graph. See Supplementary Materials D.

6. Conclusion

We presented a powerful machine learning framework for learning to simulate complex systems, based on particle-based representations of physics and learned message-passing on graphs. Our experimental results show our single GNS architecture can learn to simulate the dynamics of fluids, rigid solids, and deformable materials, interacting with one another, using tens of thousands of particles over thousands time steps. We find our model is simpler, more accurate, and has better generalization than previous approaches.

While here we focus on mesh-free particle methods, our GNS approach may also be applicable to data represented using meshes, such as finite-element methods. There are also natural ways to incorporate stronger, generic physical knowledge into our framework, such as Hamiltonian mechanics (Sanchez-Gonzalez et al., 2019) and rich, architecturally imposed symmetries. To realize advantages over traditional simulators, future work should explore how to parameterize and implement GNS computations more efficiently, and exploit the ever-improving parallel compute hardware. Learned, differentiable simulators will be valuable for solving inverse problems, by not strictly optimizing for forward prediction, but for inverse objectives as well.

More broadly, this work is a key advance toward more sophisticated generative models, and furnishes the modern AI toolkit with a greater capacity for physical reasoning.

Acknowledgements

We thank Victor Bapst, Jessica Hamrick and our reviewers for valuable feedback on the work and manuscript, and we thank Benjamin Ummenhofer for advice on implementing the continuous convolution baseline model.

References

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Battaglia, P., Pascanu, R., Lai, M., Rezende, D. J., et al. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, pp. 4502–4510, 2016.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Bender, J. and Koschier, D. Divergence-free smoothed particle hydrodynamics. In *Proceedings of the 2015 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. ACM, 2015. doi: http://dx.doi.org/10.1145/2786784.2786796.

Chang, M. B., Ullman, T., Torralba, A., and Tenenbaum, J. B. A compositional object-based approach to learning physical dynamics. *arXiv preprint arXiv:1612.00341*, 2016.

Chen, J., Fang, H.-r., and Saad, Y. Fast approximate kNN graph construction for high dimensional data via recursive lanczos bisection. *Journal of Machine Learning Research*, 10(Sep):1989–2012, 2009.

Cuturi, M. Sinkhorn distances: Lightspeed computation of optimal transportation distances, 2013.

Graph Nets Library. DeepMind, 2018. URL `https://github.com/deepmind/graph_nets`.

Dong, W., Moses, C., and Li, K. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, pp. 577–586, 2011.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1263–1272. JMLR.org, 2017.

Gretton, A., Borgwardt, K. M., Rasch, M. J., Schölkopf, B., and Smola, A. A kernel two-sample test. *Journal of Machine Learning Research*, 13(Mar):723–773, 2012.

Grzeszczuk, R., Terzopoulos, D., and Hinton, G. Neuroanimator: Fast neural network emulation and control of physics-based models. In *Proceedings of the 25th Annual Conference on Computer graphics and Interactive Techniques*, pp. 9–20, 1998.

He, S., Li, Y., Feng, Y., Ho, S., Ravanbakhsh, S., Chen, W., and Póczos, B. Learning to predict the cosmological structure formation. *Proceedings of the National Academy of Sciences*, 116(28):13825–13832, 2019.

Holl, P., Koltun, V., and Thuerey, N. Learning to control PDEs with differentiable physics. *arXiv preprint arXiv:2001.07457*, 2020.

Hu, Y., Fang, Y., Ge, Z., Qu, Z., Zhu, Y., Pradhana, A., and Jiang, C. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph.*, 37(4), July 2018.

Hu, Y., Anderson, L., Li, T.-M., Sun, Q., Carr, N., Ragan-Kelley, J., and Durand, F. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Kumar, S., Bitorff, V., Chen, D., Chou, C., Hechtman, B., Lee, H., Kumar, N., Mattson, P., Wang, S., Wang, T., et al. Scale mlperf-0.6 models on google tpu-v3 pods. *arXiv preprint arXiv:1909.09756*, 2019.

Ladický, L., Jeong, S., Solenthaler, B., Pollefeys, M., and Gross, M. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (TOG)*, 34(6): 1–9, 2015.

Li, Y., Wu, J., Tedrake, R., Tenenbaum, J. B., and Torralba, A. Learning particle dynamics for manipulating rigid bodies, deformable objects, and fluids. *arXiv preprint arXiv:1810.01566*, 2018.

Li, Y., Wu, J., Zhu, J.-Y., Tenenbaum, J. B., Torralba, A., and Tedrake, R. Propagation networks for model-based control under partial observation. In *2019 International Conference on Robotics and Automation (ICRA)*, pp. 1205–1211. IEEE, 2019.

Macklin, M., Müller, M., Chentanez, N., and Kim, T.-Y. Unified particle physics for real-time applications. *ACM Transactions on Graphics (TOG)*, 33(4):1–12, 2014.

Manessi, F., Rozza, A., and Manzo, M. Dynamic graph convolutional networks. *Pattern Recognition*, 97:107000, 2020.

Monaghan, J. J. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30(1):543–574, 1992.

Mrowca, D., Zhuang, C., Wang, E., Haber, N., Fei-Fei, L. F., Tenenbaum, J., and Yamins, D. L. Flexible neural representation for physics prediction. In *Advances in Neural Information Processing Systems*, pp. 8799–8810, 2018.

Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.

Sanchez-Gonzalez, A., Heess, N., Springenberg, J. T., Merel, J., Riedmiller, M., Hadsell, R., and Battaglia, P. Graph networks as learnable physics engines for inference and control. *arXiv preprint arXiv:1806.01242*, 2018.

Sanchez-Gonzalez, A., Bapst, V., Cranmer, K., and Battaglia, P. Hamiltonian graph networks with ode integrators. *arXiv preprint arXiv:1909.12790*, 2019.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

Schoenholz, S. S. and Cubuk, E. D. Jax, md: End-to-end differentiable, hardware accelerated, molecular dynamics in pure python. *arXiv preprint arXiv:1912.04232*, 2019.

Sulsky, D., Zhou, S.-J., and Schreyer, H. L. Application of a particle-in-cell method to solid mechanics. *Computer physics communications*, 87(1-2):236–252, 1995.

Sun, C., Karlsson, P., Wu, J., Tenenbaum, J. B., and Murphy, K. Stochastic prediction of multi-agent interactions from partial observations. *arXiv preprint arXiv:1902.09641*, 2019.

Tacchetti, A., Song, H. F., Mediano, P. A., Zambaldi, V., Rabinowitz, N. C., Graepel, T., Botvinick, M., and Battaglia, P. W. Relational forward models for multi-agent learning. *arXiv preprint arXiv:1809.11044*, 2018.

Tang, J., Liu, J., Zhang, M., and Mei, Q. Visualizing large-scale and high-dimensional data. In *Proceedings of the 25th International Conference on World Wide Web*, pp. 287–297, 2016.

Trivedi, R., Dai, H., Wang, Y., and Song, L. Know-evolve: Deep temporal reasoning for dynamic knowledge graphs. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 3462–3471. JMLR. org, 2017.

Trivedi, R., Farajtabar, M., Biswal, P., and Zha, H. Dyrep: Learning representations over dynamic graphs. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=HyePrhR5KX`.

Ummenhofer, B., Prantl, L., Thürey, N., and Koltun, V. Lagrangian fluid simulation with continuous convolutions. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=B1lDoJSYDH`.

Veličković, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=SkgKO0EtvS`.

Villani, C. *Topics in optimal transportation*. American Mathematical Soc., 2003.

Wiewel, S., Becher, M., and Thuerey, N. Latent space physics: Towards learning the temporal evolution of fluid flow. In *Computer Graphics Forum*, pp. 71–82. Wiley Online Library, 2019.

Yan, S., Xiong, Y., and Lin, D. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *Thirty-second AAAI Conference on Artificial Intelligence*, 2018.