

DeepMind

Deep Dive on Graph Networks for Learning Simulation

4th Inter-experiment Machine Learning Workshop

22 October 2020

Alvaro Sanchez-Gonzalez – DeepMind

Victor Bapst, Peter Battaglia, Kyle Cranmer, Miles Cranmer,
Meire Fortunato, Jonathan Godwin, Jessica Hamrick,
Shirley Ho, Jure Leskovec, Tobias Pfaff, Rex Ying,

DeepMind 

Simulation is fundamental to science and technology

Largest supercomputers in the world (Nov 2019)

#1. "Summit" @ Oak Ridge: "[A Sneak Peek at 19 Science Simulations for the Summit Supercomputer in 2019](#)"

1. *Evolution of the universe*
2. *Whole-cell simulation*
3. *Inside a nuclear reactor*
4. *Post-Moore's Law graphene circuits*
5. *Formation of matter*
6. *Cell's molecular machine*
7. *Unpacking the nucleus*
8. *Mars landing*
9. *Deep learning for microscopy*
10. *Elements from star explosions*
11. *Cancer data*
12. *Earthquake resilience for cities*
13. *Nature of elusive neutrinos*
14. *Extreme weather with deep learning*
15. *Flexible, lightweight solar cells*
16. *Virtual fusion reactor*
17. *Unpredictable material properties*
18. *Genetic clues in the opioid crisis*
19. *Turbulent environments*



Simulation is fundamental to science and technology

Largest supercomputers in the world (Nov 2019)

#1 "Summit" @ Oak Ridge: "A Sneak Peek at 19 Science **Simulations** for the Summit Supercomputer in 2019"

#2 "Sierra" @ Lawrence Livermore: "[nuclear] **simulation** in lieu of underground testing"

#3 "Sunway TaihuLight" @ NSC, Wuxi: "**simulated** the Universe with 10 trillion digital particles"

#4 "Tianhe-2A" @ NSC, Guangzhou: "main application ... is for computational fluid dynamics (CFD) ... aircraft **simulations**"

#5 "Frontera" @ TACC: "high-resolution climate **simulations**, molecular dynamics models with millions of atoms"

#6 "Piz Daint" @ CSCS: "**simulate** processes for projects in geophysics, materials science, chemistry, ... climate modeling"

#7 "Trinity" @ Los Alamos: "A trillion-particle **simulation**? No sweat for the Trinity supercomputer at Los Alamos"

(#8 "ABCI" @ AIST, Japan: not simulation, but deep learning)

#9 "SuperMUC-NG" @ Leibniz Supercomputing Centre: "Researchers Visualize the Largest Turbulence **Simulation** Ever"

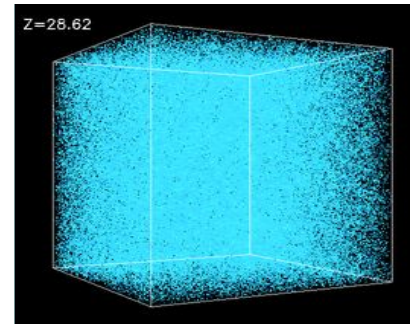
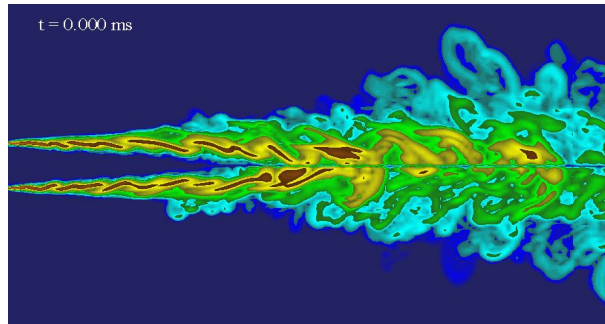
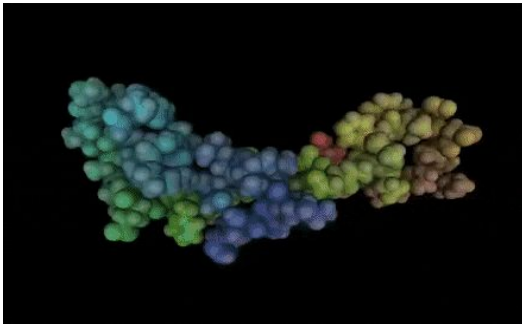
#10 "Lassen" @ Lawrence Livermore: "The system is designated for unclassified **simulation** and analysis"



Why *learn* simulation?

Engineered simulators:

1. Substantial effort to build
2. Substantial resources to run
3. Only as accurate as the designer
4. Not always suitable for solving inverse problems



Why *learn* simulation?

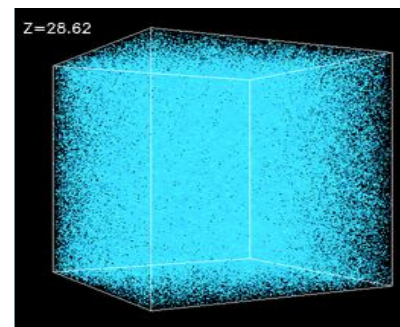
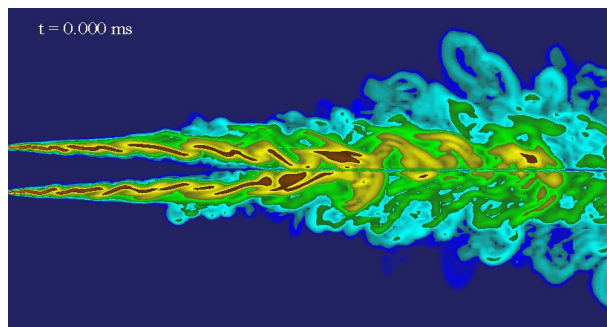
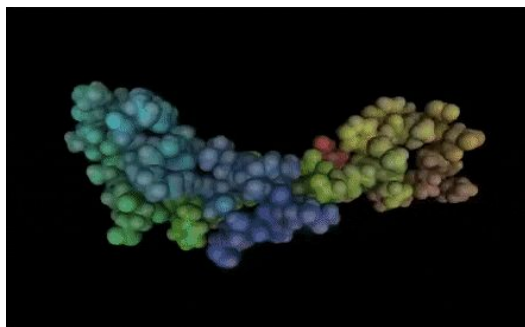
Engineered simulators:

1. Substantial effort to build
2. Substantial resources to run
3. Only as accurate as the designer
4. Not always suitable for solving inverse problems



Learned simulators:

1. Shared architectures
2. Accuracy–efficiency trade off
3. As accurate as the available data
4. Gradient–based planning
5. Interpretable models!*



*[“Discovering Symbolic Models from Deep Learning with Inductive Biases”](#)



Graph Networks for Learning Physical Simulation

- **Deep Dive** on our most recent models:

“Learning to Simulate Complex Physics with Graph Networks” (ICML 2020)

Alvaro Sanchez-Gonzalez, Jonathan Godwin*, Tobias Pfaff*, et al.*

Arxiv: arxiv.org/abs/2002.09405

Video page: sites.google.com/view/learning-to-simulate

“Learning Mesh-Based Simulation with Graph Networks” (arXiv, under review)

Tobias Pfaff, Meire Fortunato*, Alvaro Sanchez-Gonzalez*, Peter Battaglia*

Arxiv: arxiv.org/abs/2010.03409

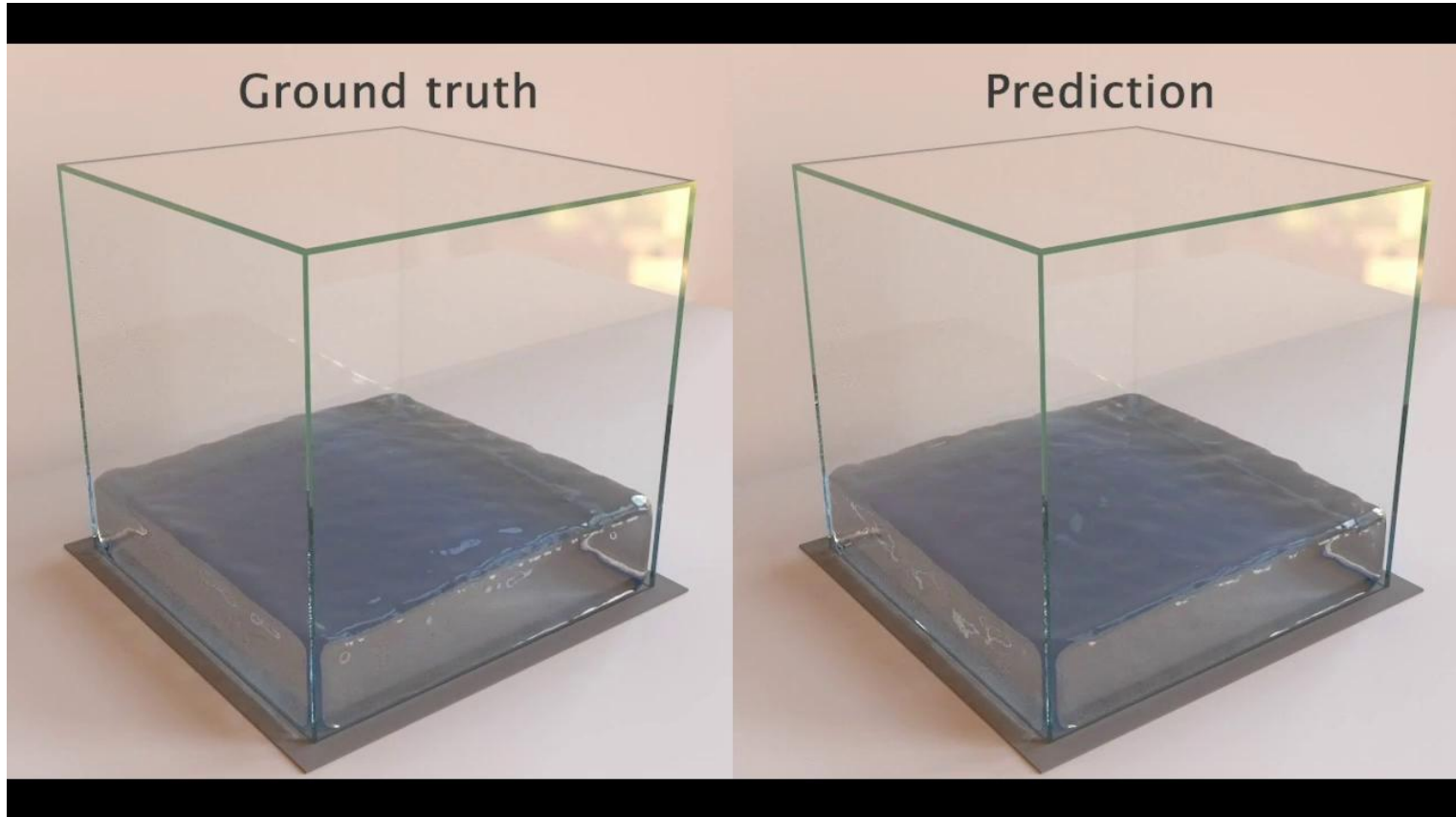
Video page: sites.google.com/view/meshgraphnets

- Focus on **general principles behind the design**

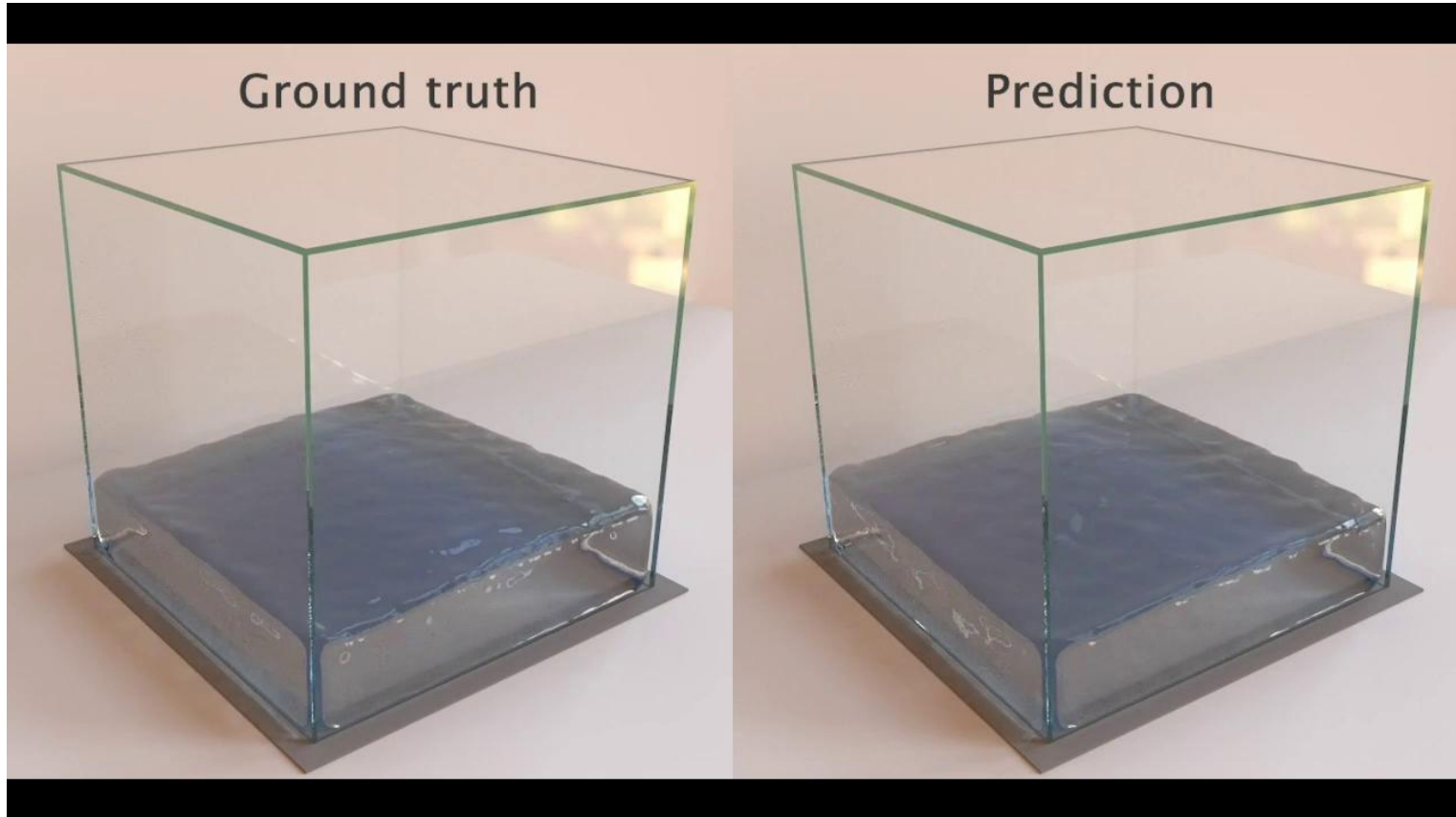
- Applicable to other domains



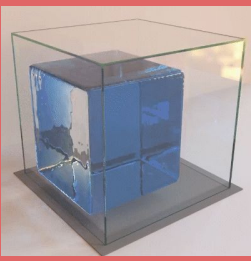
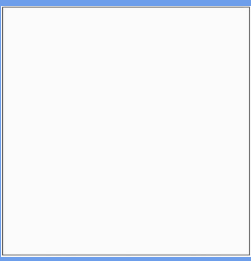
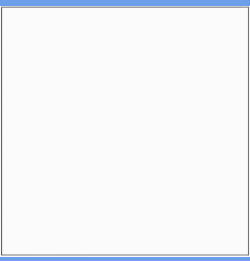
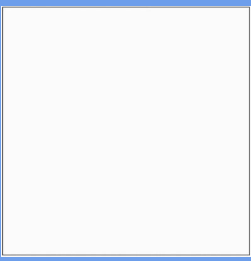

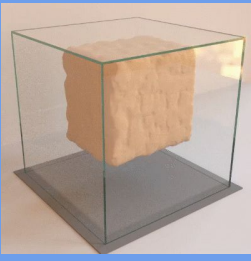
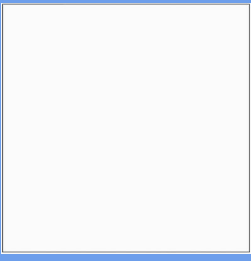
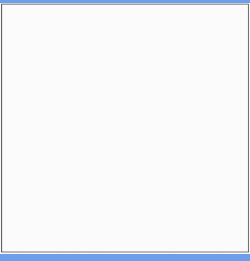
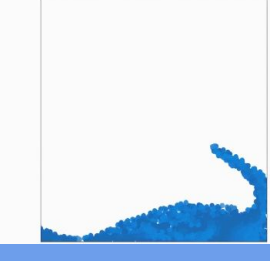


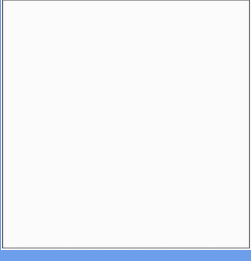
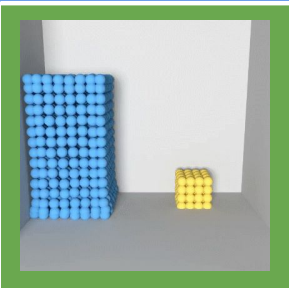
Water simulation (SPH)



Water simulation (SPH)



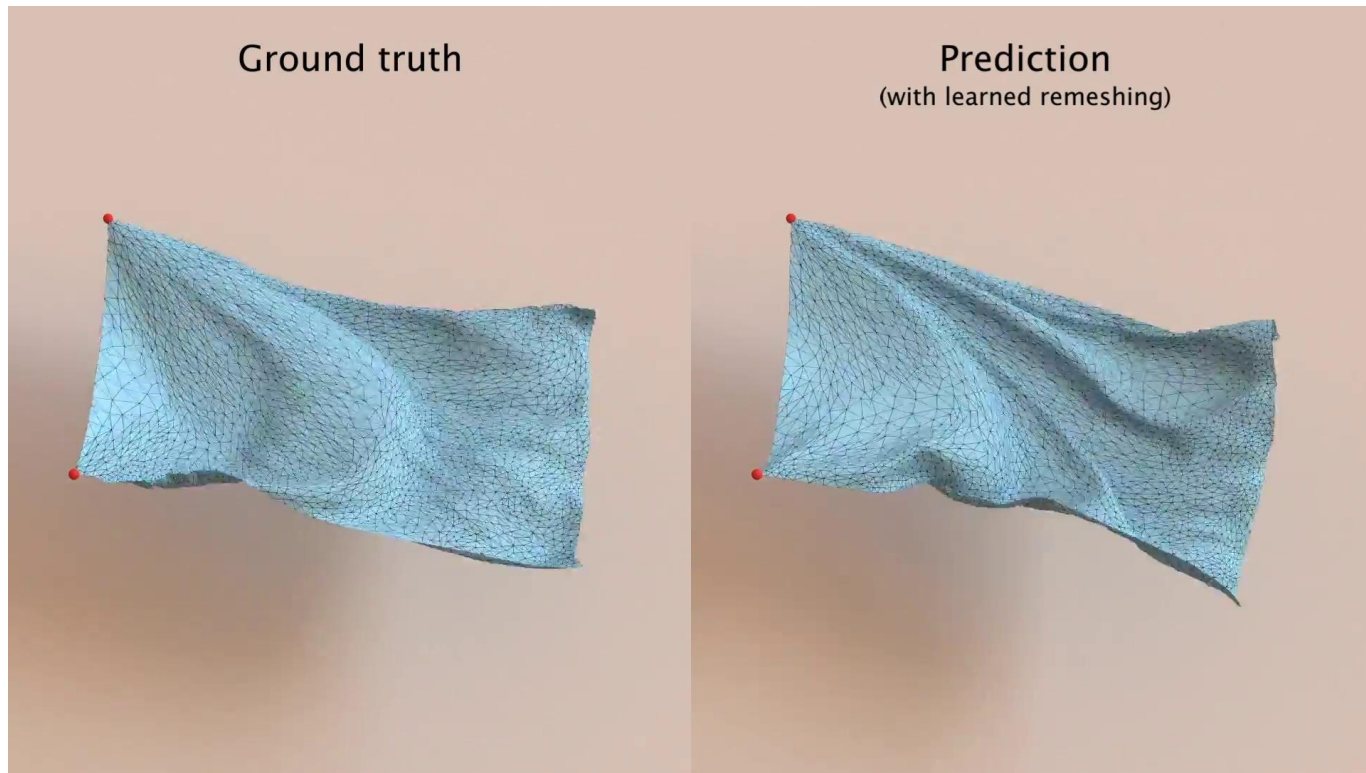
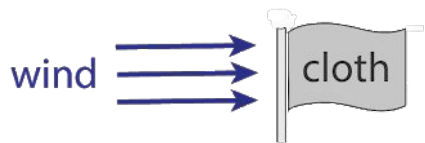
Multiple materials

				
				
			<p><u>Data from 3 distinct simulators:</u></p> <p>SPH MPM PBD</p>	



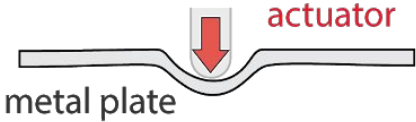
Cloth simulation (ArcSim)

- Triangular dynamic mesh
- Lagrangian representation

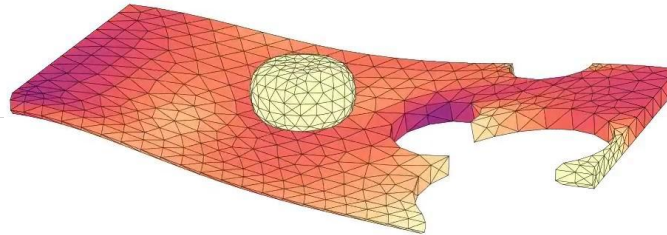


Structural dynamics (COMSOL)

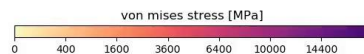
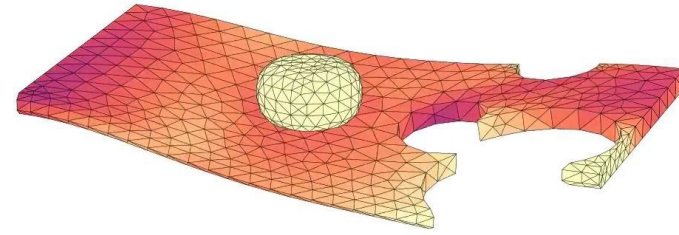
- Tetrahedral mesh
- Lagrangian representation
- Quasi-static simulation



Ground truth

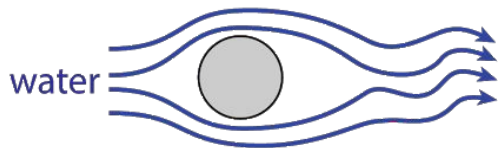


Prediction

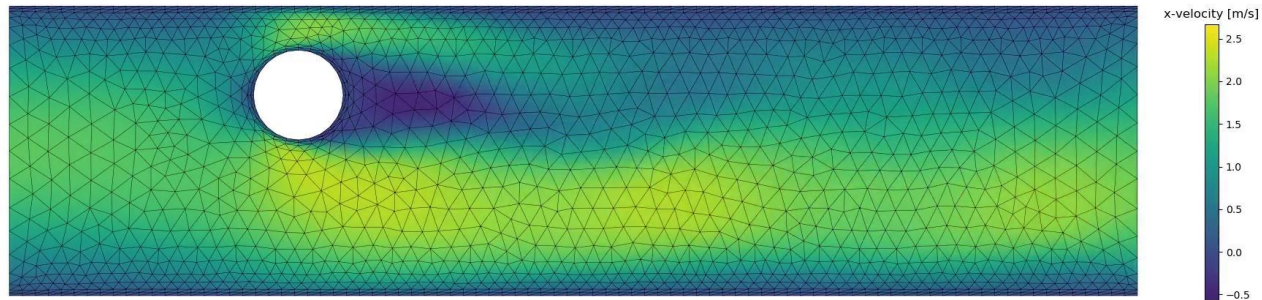


Incompressible fluids (COMSOL)

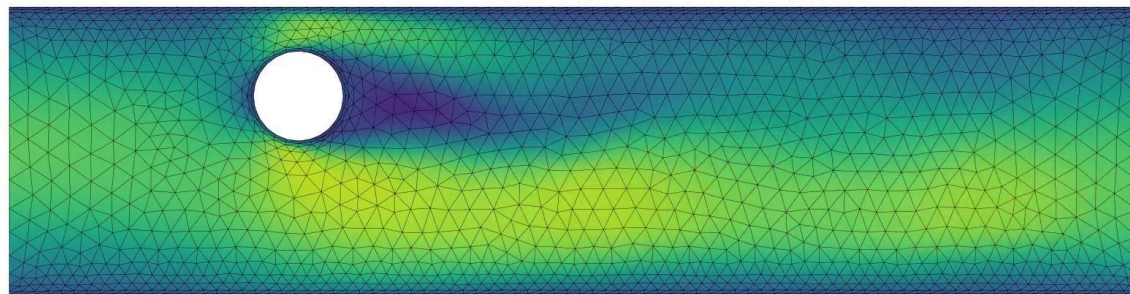
- Navier-Stokes
- Eulerian representation



Ground truth

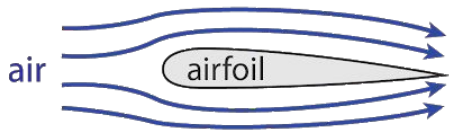


Prediction



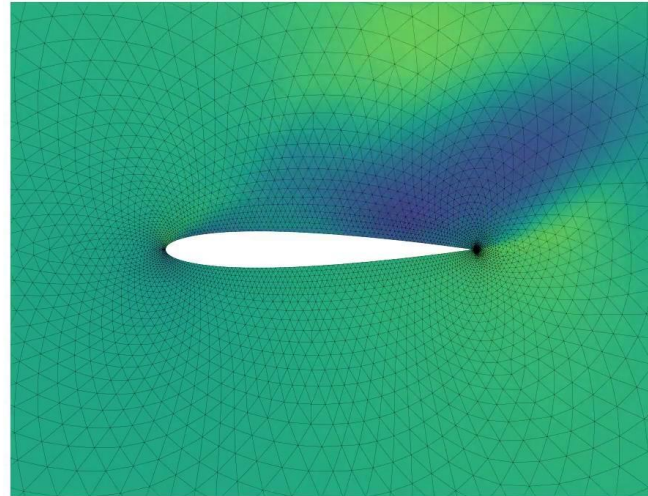
Aerodynamics (SU2)

- Navier-Stokes
- Eulerian representation

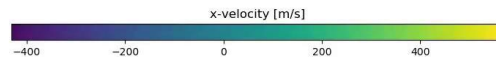
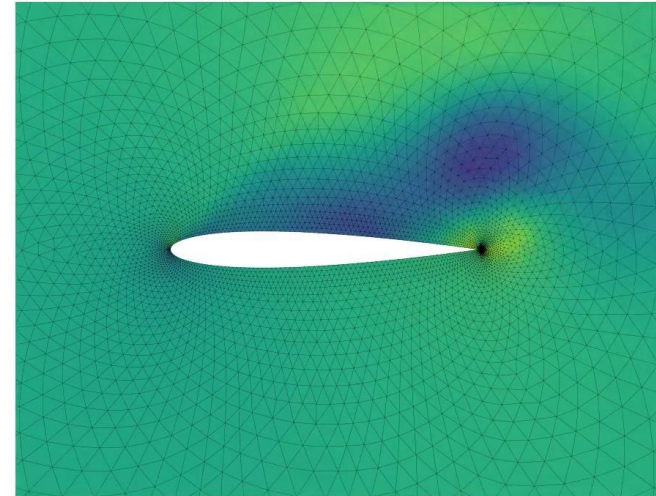


Ground truth

mach number 0.58
angle of attack 21.9



Prediction



Why Graph Network based simulators?

- **Adaptability**
 - Same model → Vastly different materials and domains
- **Data efficiency**
 - < 1000 training trajectories
- **Performance**
 - Latest model: ~10 to 100 times faster than ground truth simulator
- **Generalization**

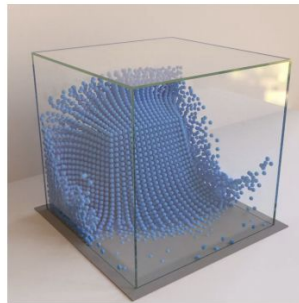


Generalization to more time-steps

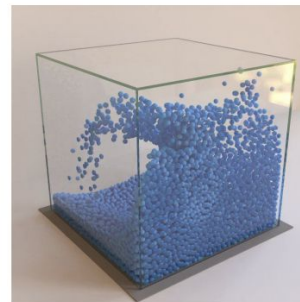
Train time

- Pairs of states

Input state

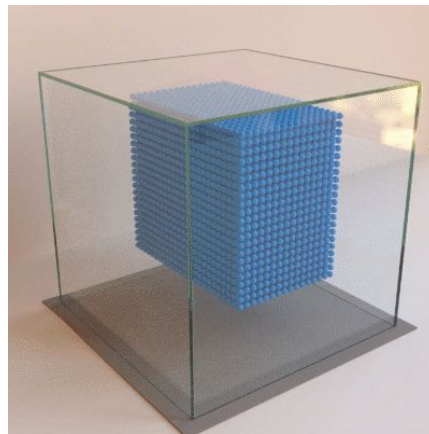


Target state



Test time

- 1000s of steps



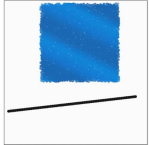
Generalization to many more particles

Training

1 x 1 domain

2k particles

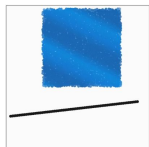
600 steps



Generalization to many more particles

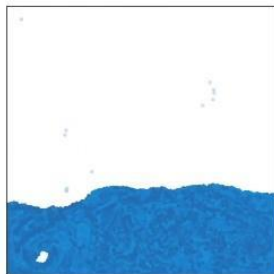
Training

1 x 1 domain
2k particles
600 steps



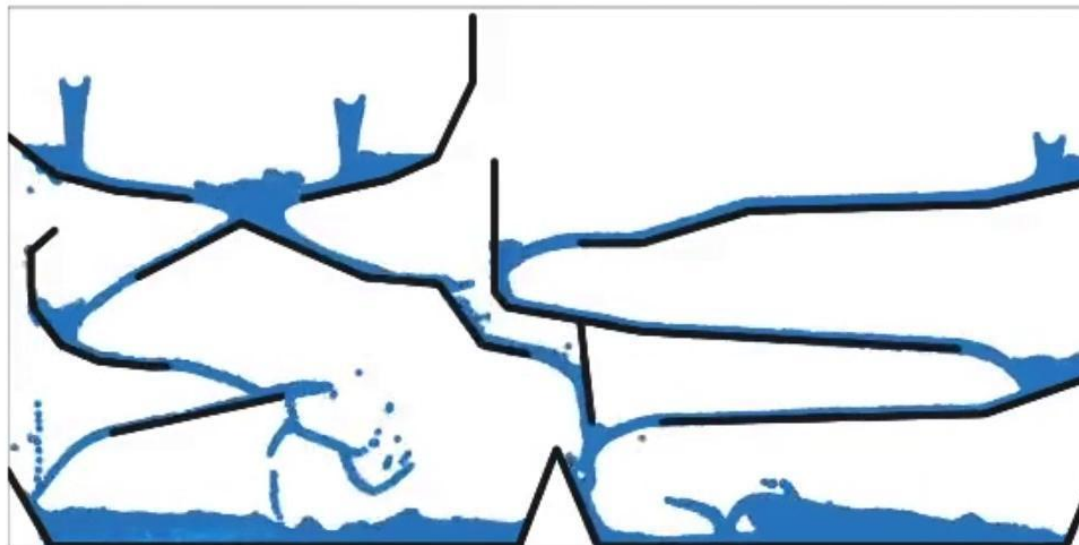
Generalization

2 x 2 domain
28k particles
2500 steps



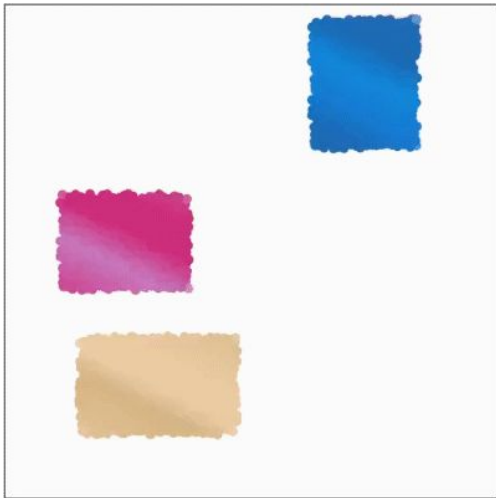
Generalization

8 x 4 domain
85k particles
5000 steps



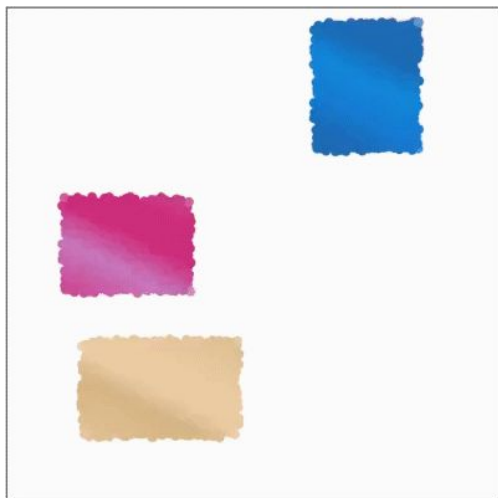
Generalization to initial conditions

Training



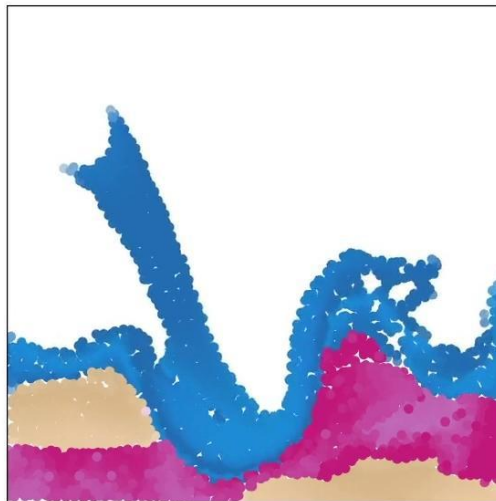
Generalization to initial conditions

Training

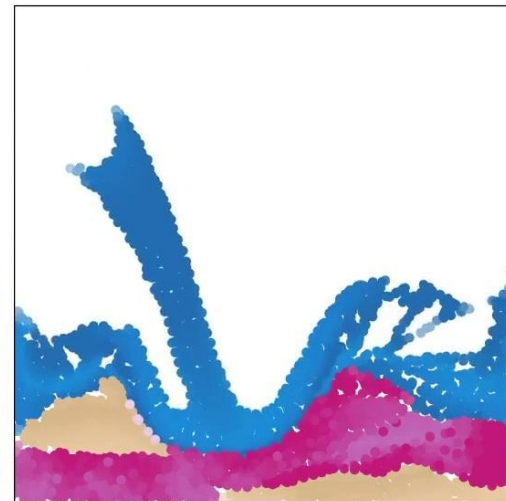


Generalization

Ground truth



Prediction

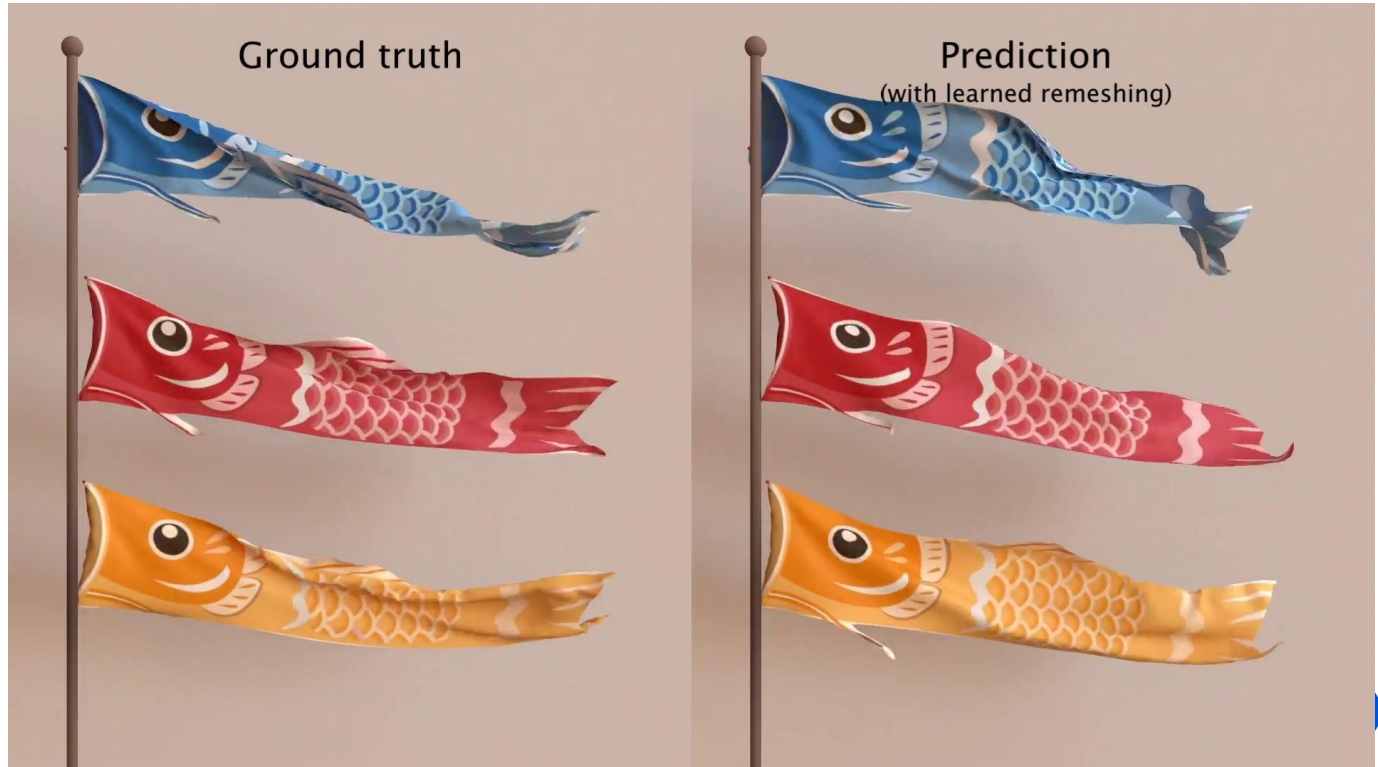
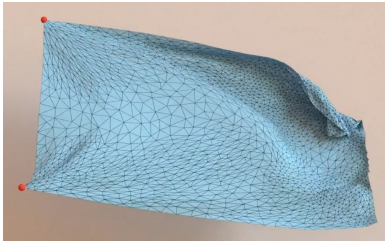


slow motion

Generalization to different meshes

Generalization

Training



Generalization to larger meshes

Ground truth



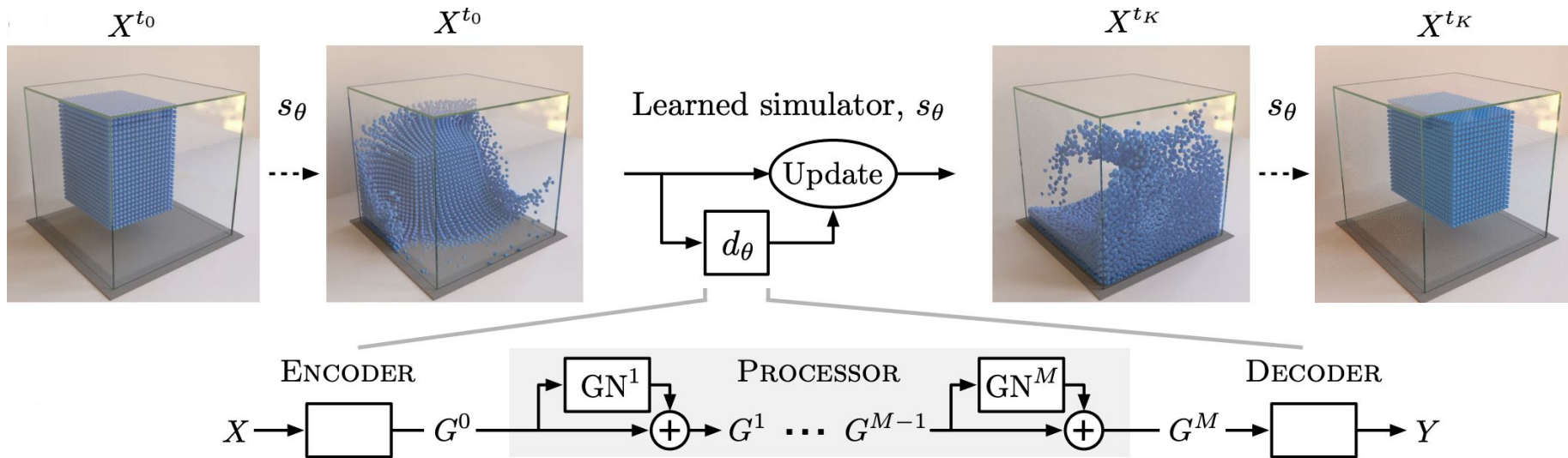
Prediction
(with learned remeshing)



Training:
2k nodes

Generalization:
>20k nodes

Graph Network-Based Simulators



Main design principle: **neural networks are dumb, let's make their life easy**



*“If I have seen further it is by
standing on the shoulders of giants.”*

-Sir Isaac Newton-

Our Neural Networks should also *have
the knowledge of giants!*

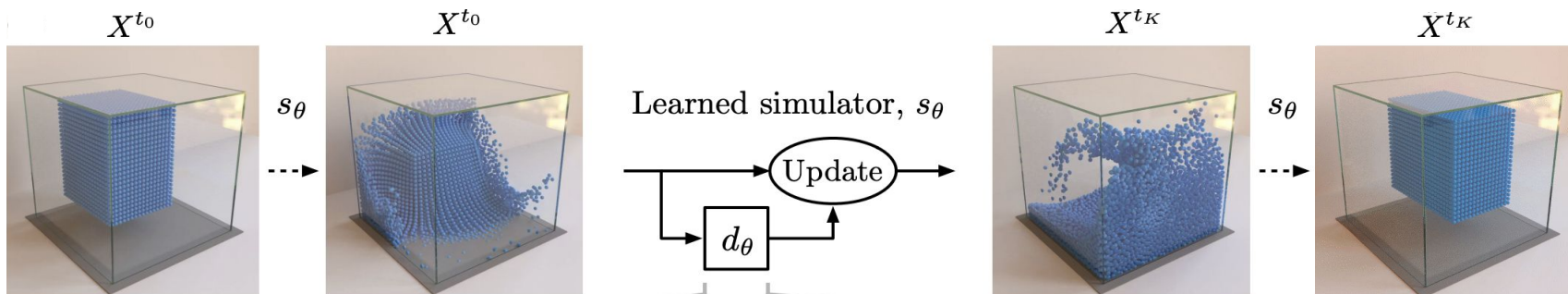


Inductive Biases

Physics-inspired inductive biases

“An **inductive bias** allows a learning algorithm to prioritize one solution (or interpretation) over another.”

Mitchell, T. M.. *The need for biases in learning generalizations.* (1980)



**Spatial
equivariance**

**Local
interactions**

**Universal
rules**

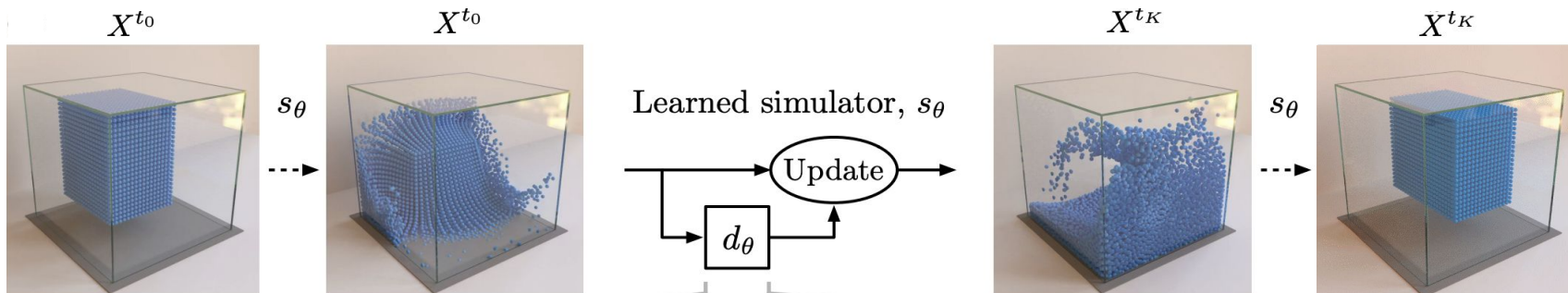
**Pairwise
interactions
Permutation equivariance**

**Superposition
principle**

**Differential
equations**



Physics-inspired inductive biases



**Spatial
equivariance**

**Local
interactions**

**Universal
rules**

**Pairwise
interactions**
Permutation equivariance

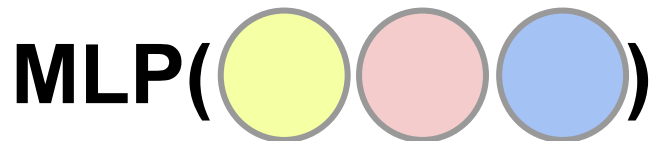
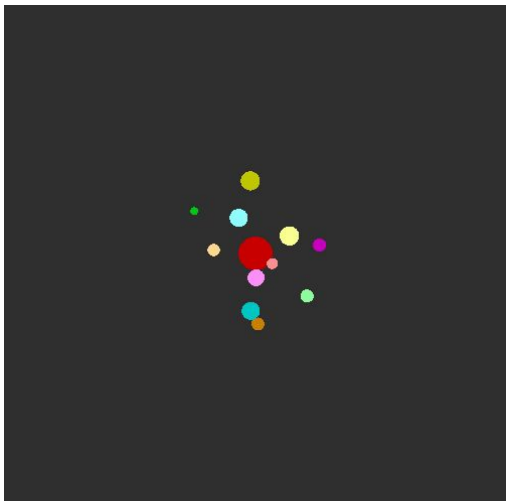
**Superposition
principle**

**Differential
equations**



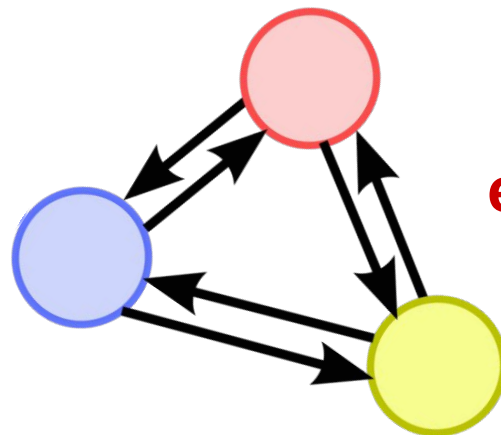
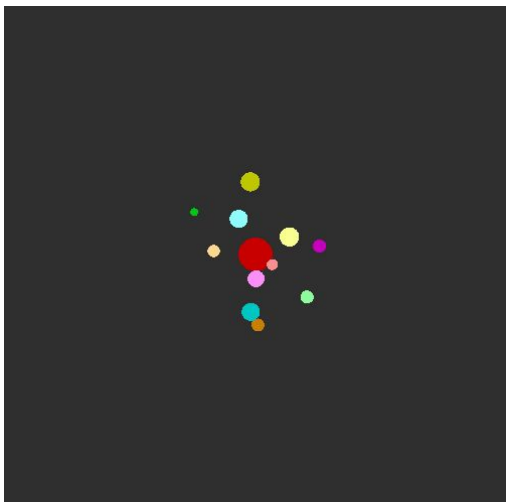
Graph Networks

- MLPs operate over vectors



Graph Networks

- Neural networks that **operate over graphs**

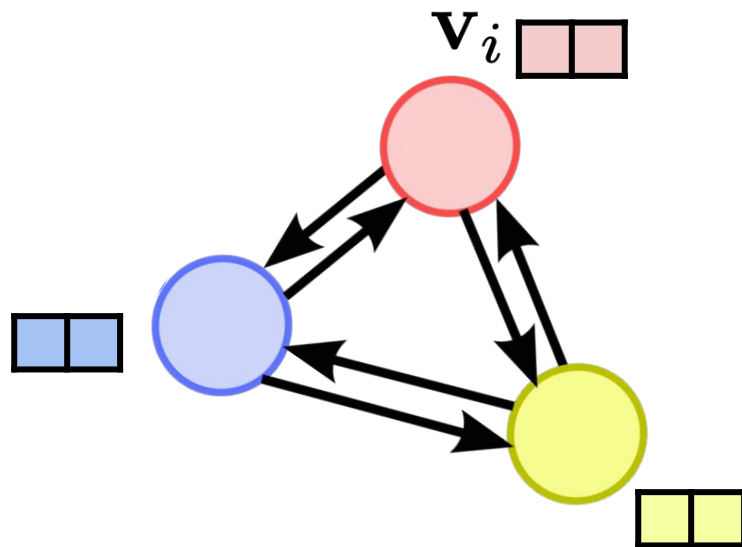
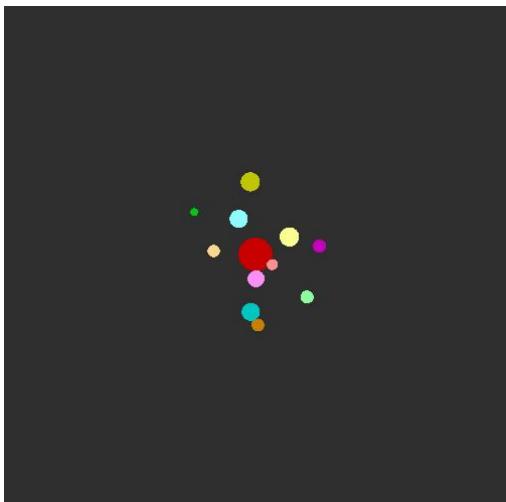


**Permutation
equivariance**



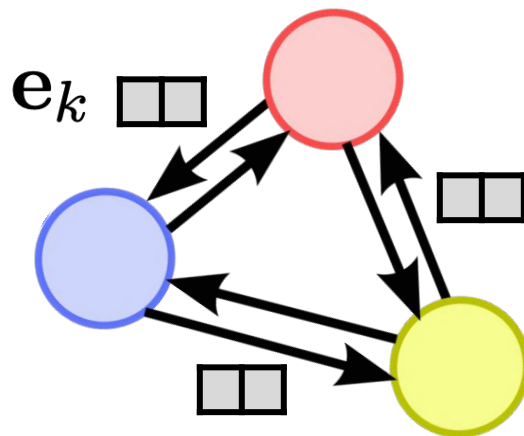
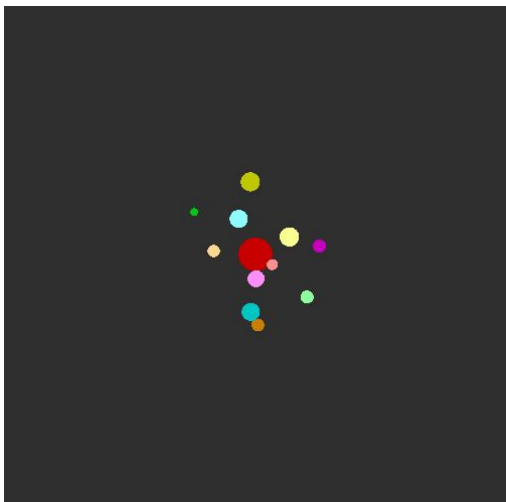
Graph Networks

- Neural networks that **operate over graphs**
 - Node features



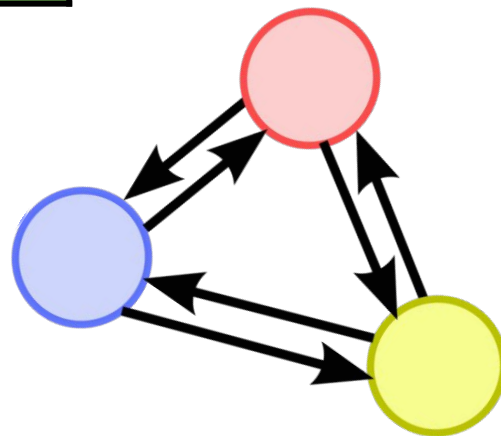
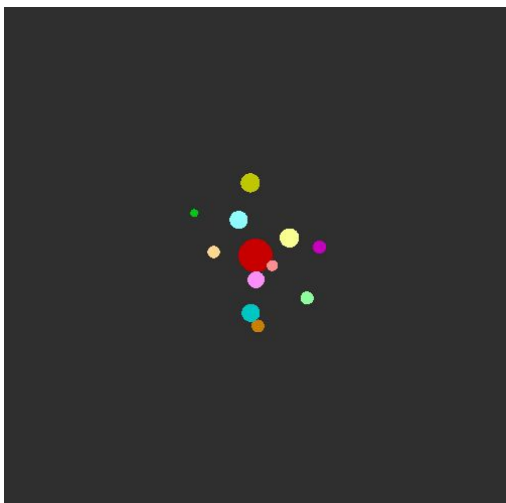
Graph Networks

- Neural networks that **operate over graphs**
 - Edge features



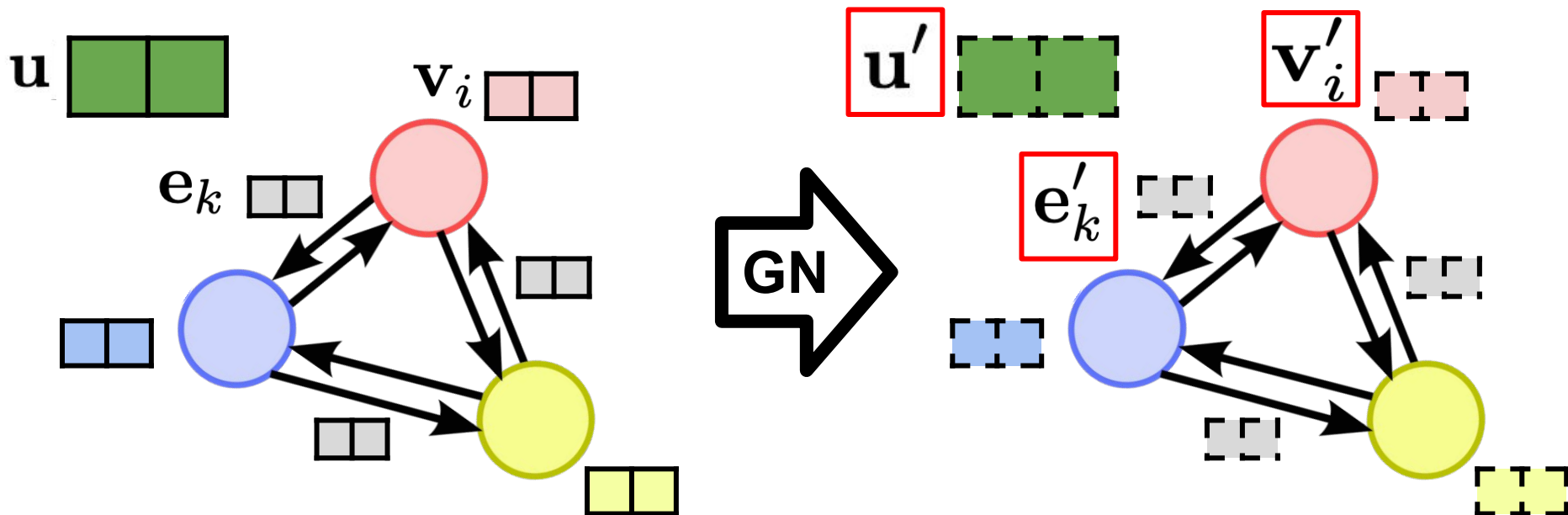
Graph Networks

- Neural networks that **operate over graphs**
 - Global features



Graph Networks

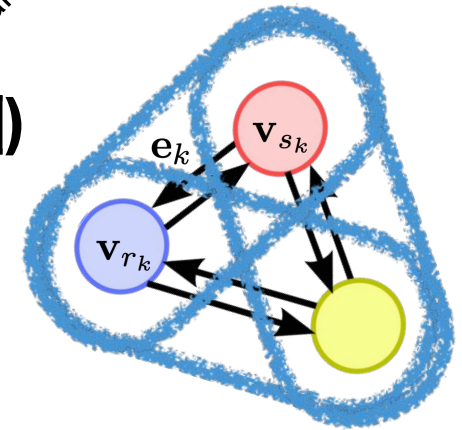
Update edge, node and global embeddings



Message passing: Edge update

Edge (message) function (for every edge)

$$\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}) := \text{NN}_e \left(\begin{array}{|c|c|c|c|c|c|c|} \hline \text{Edge features} & \text{Receiver node features} & \text{Sender node features} & \text{Global features} & & & \\ \hline \end{array} \right)$$



**Pairwise
interactions**

**Universal
rules**



Message passing: Node update

Receiver edge aggregation (Message pooling) (for every node)

$$\bar{\mathbf{e}}'_i \leftarrow \sum_{r_k=i} \mathbf{e}'_k$$

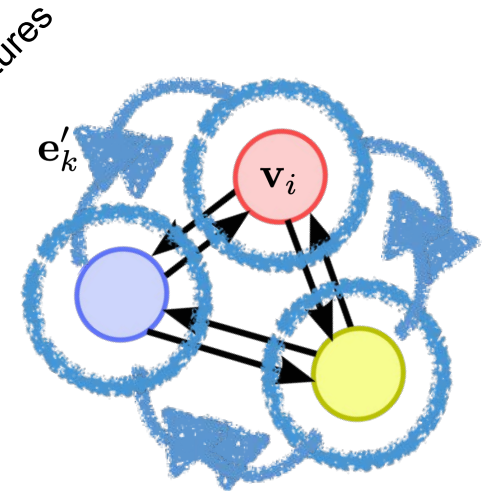
Superposition principle

Node function (for every node)

$$\mathbf{v}'_i \leftarrow \phi^v (\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}) := \text{NN}_v (\text{Aggregated edge features} \quad \text{Node features} \quad \text{Global features})$$

Local interactions

Universal rules



Graph Networks: Global update

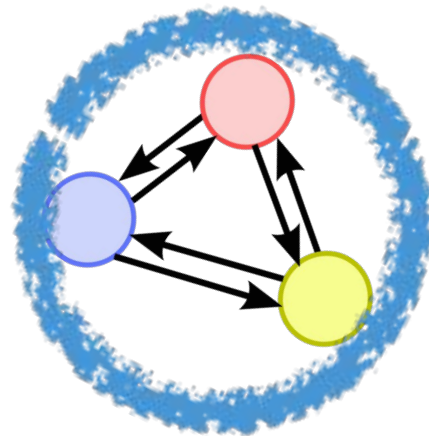
Global node and edge aggregation

$$\bar{\mathbf{v}}' \leftarrow \sum_i \mathbf{v}'_i \quad \bar{\mathbf{e}}' \leftarrow \sum_k \mathbf{e}'_k$$

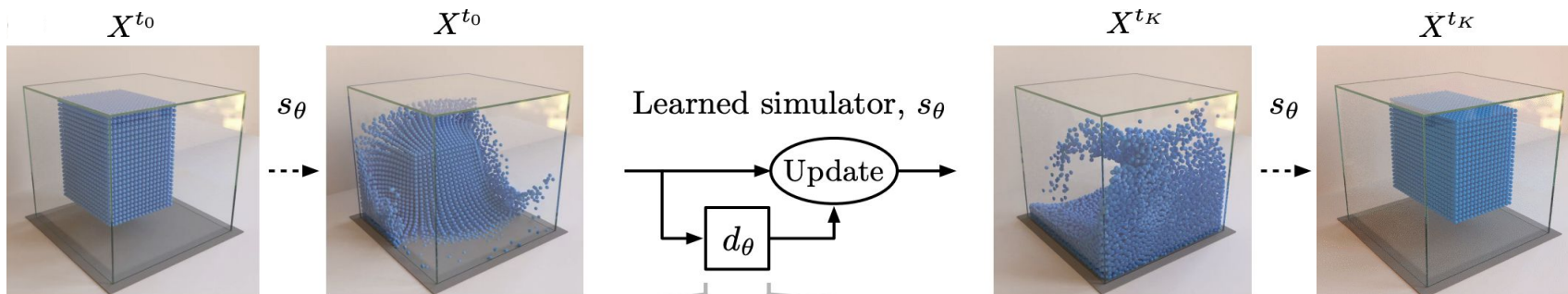
Global function

$$\mathbf{u}' \leftarrow \phi^u (\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}) := \text{NN}_u \left(\begin{array}{|c|c|c|c|} \hline \text{Aggregated edge features} & \text{Aggregated node features} & \text{Global features} & \text{Global features} \\ \hline \end{array} \right)$$

~~Local interactions~~



Physics-inspired inductive biases



**Spatial
equivariance**

**Local
interactions**

**Universal
rules**

**Pairwise
interactions**
**Superposition
principle**
Permutation equivariance

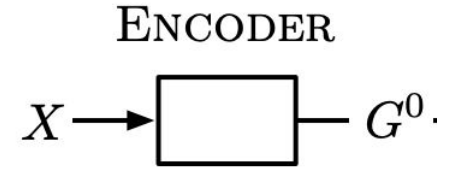
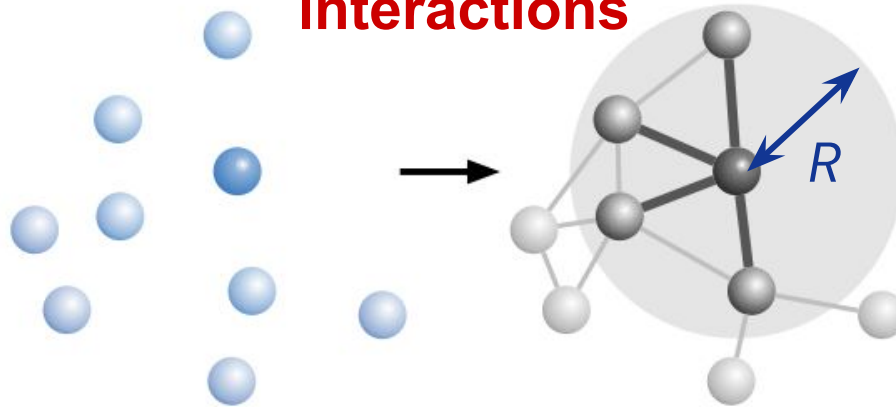
**Differential
equations**



Encoder

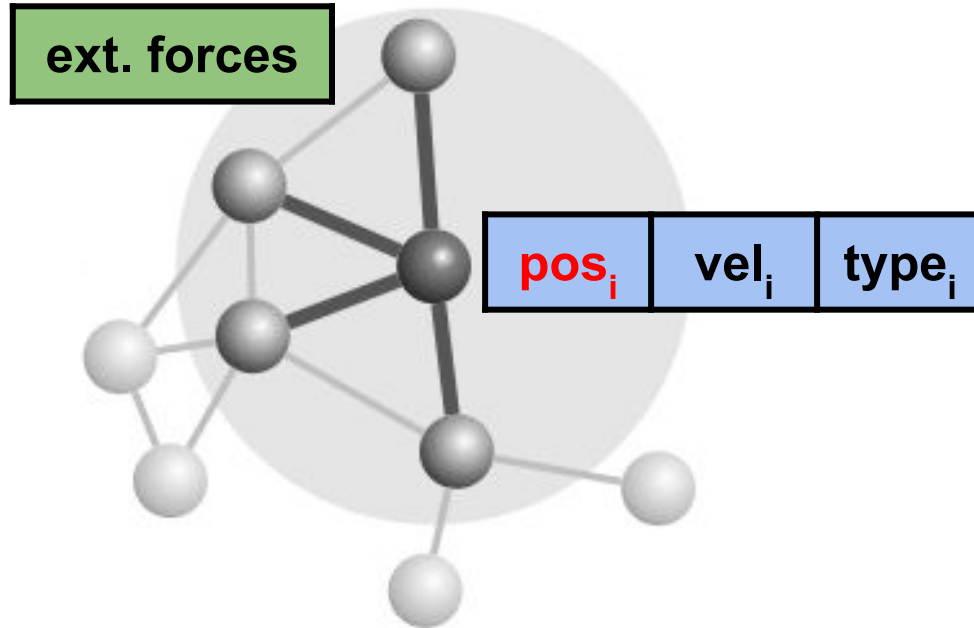
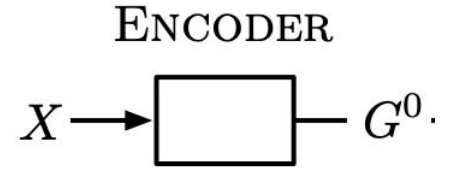
- Transform the inputs into a graph
 - Add connectivity with a certain radius R

**Local
(spatial)
interactions**



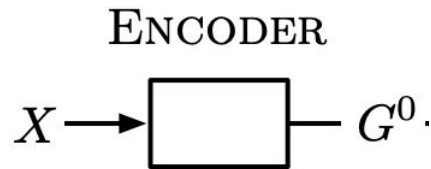
Encoder

- Add features to the graph (naive approach)

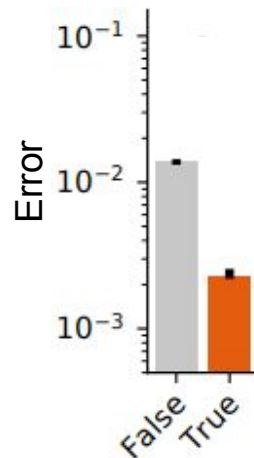
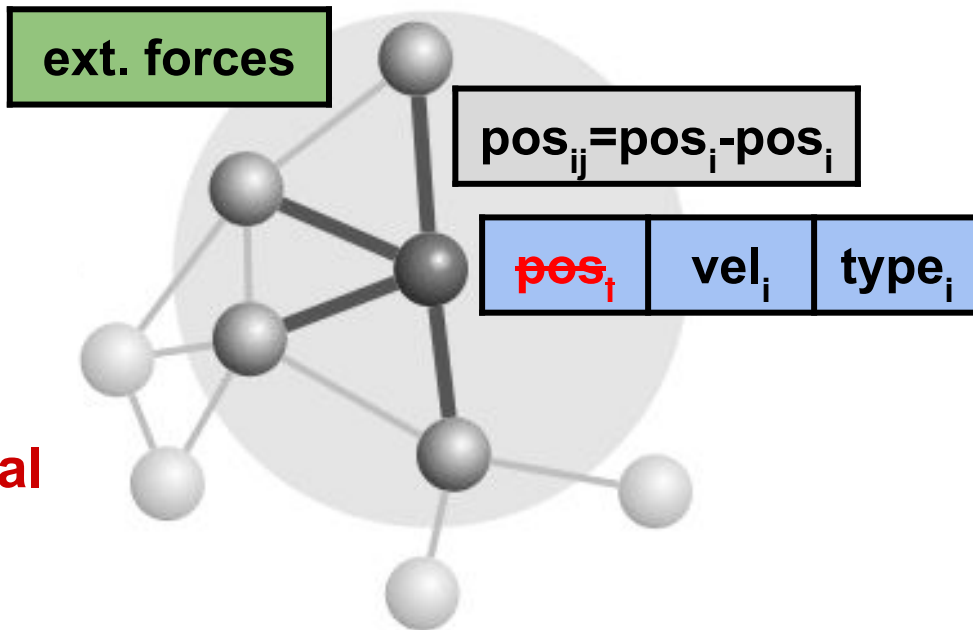


Encoder

- Add features to the graph



Relative positional features



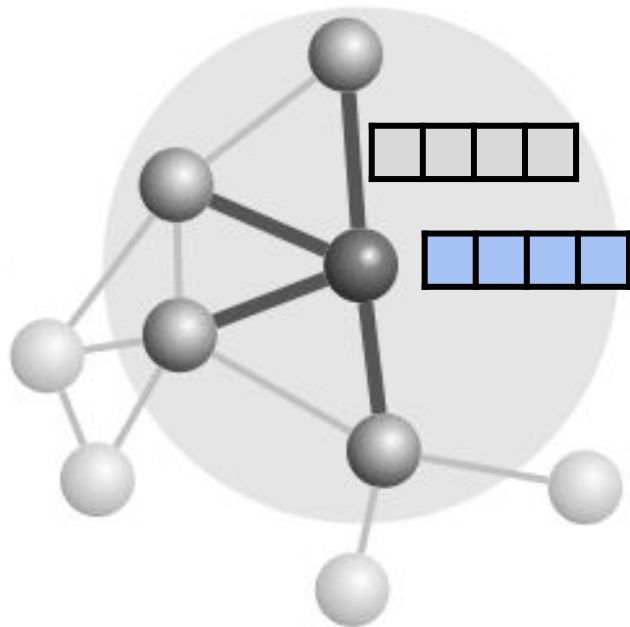
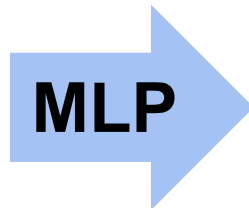
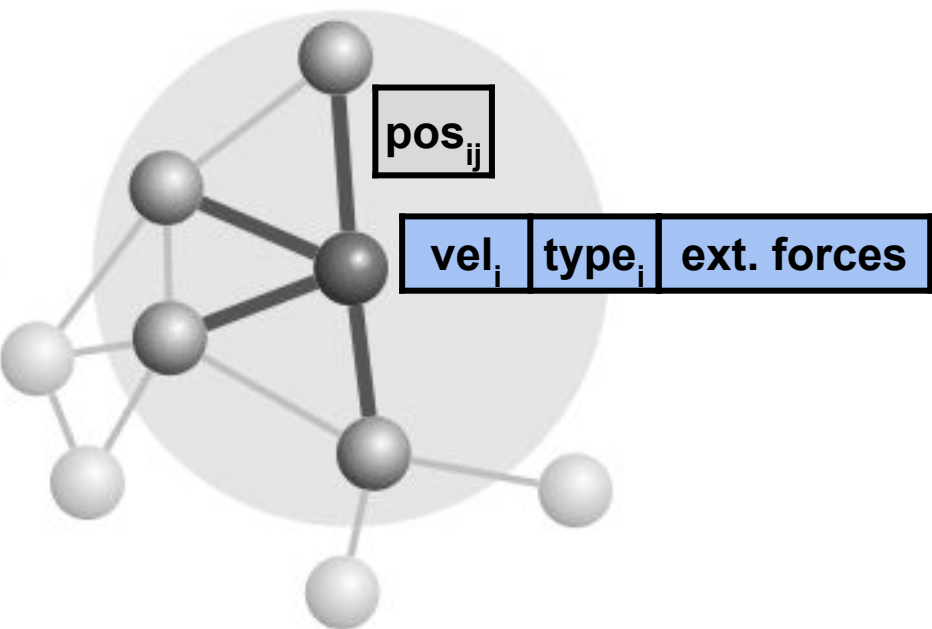
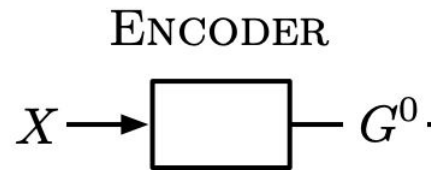
Relative positional features



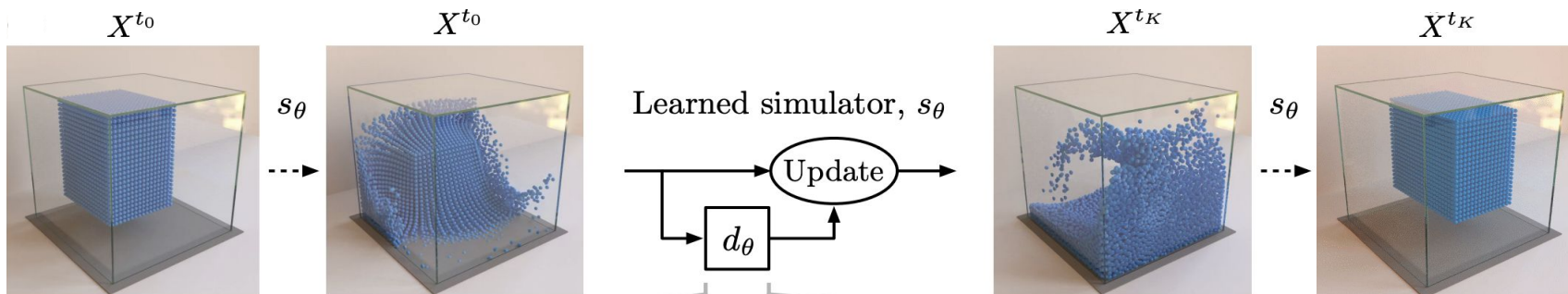
Spatial equivariance **Universal rules**

Encoder

- Embed graph features



Physics-inspired inductive biases



**Spatial
equivariance**

**Local
interactions**

**Universal
rules**

**Pairwise
interactions**
Permutation equivariance

**Superposition
principle**

**Differential
equations**



Processor: Graph Network stack

- Iterative message passing (MP) without global updates

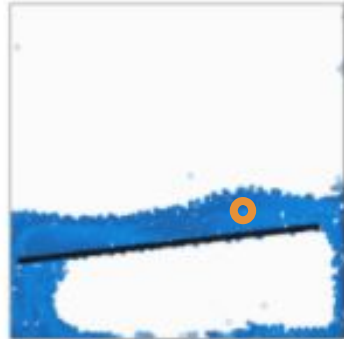


- Increase range of communication

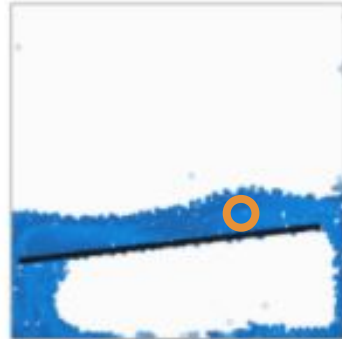
After 1 MP steps



After 2 MP steps



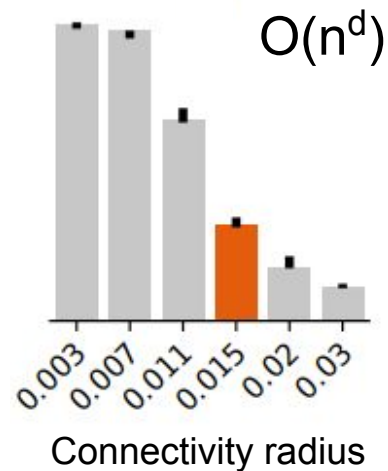
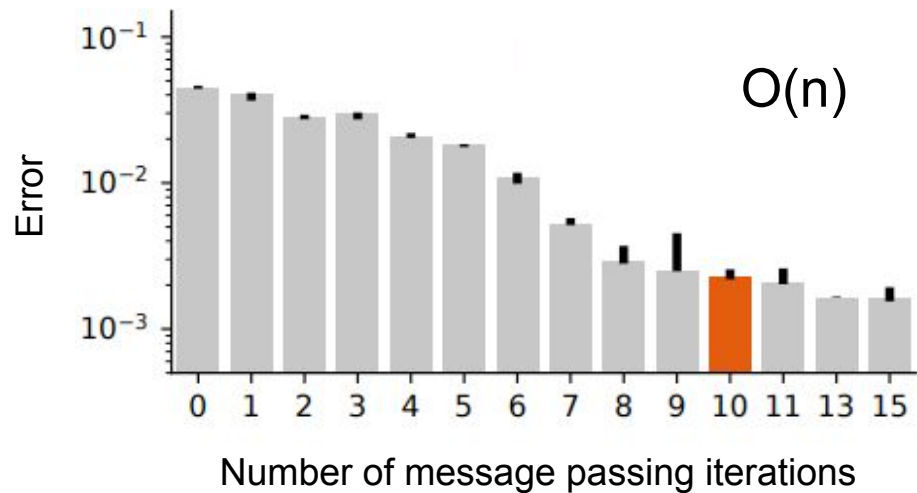
After 3 MP steps



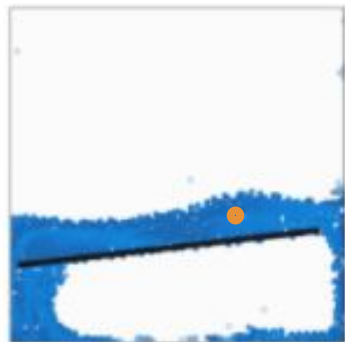
After n MP steps



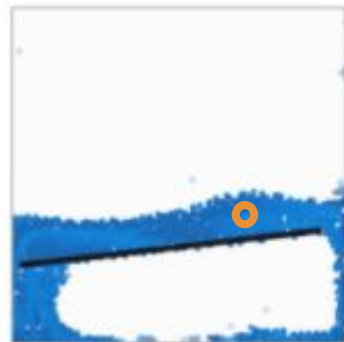
Local interactions



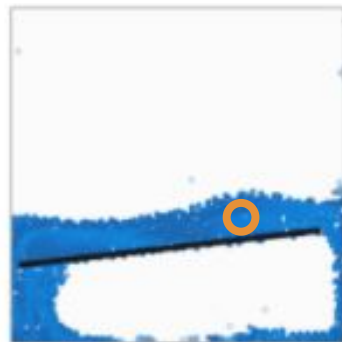
After 1 MP steps



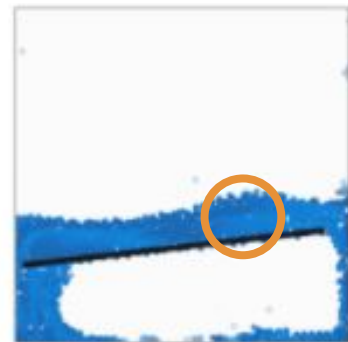
After 2 MP steps



After 3 MP steps

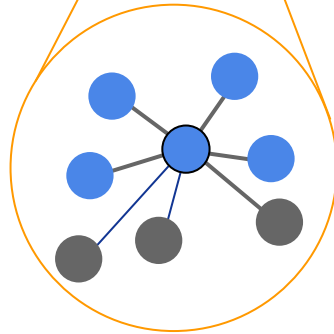
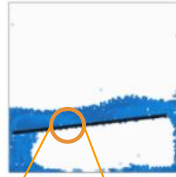


After n MP steps



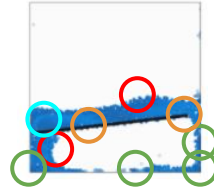
How does generalization work?

Training



Generalization to significantly more particles

Training



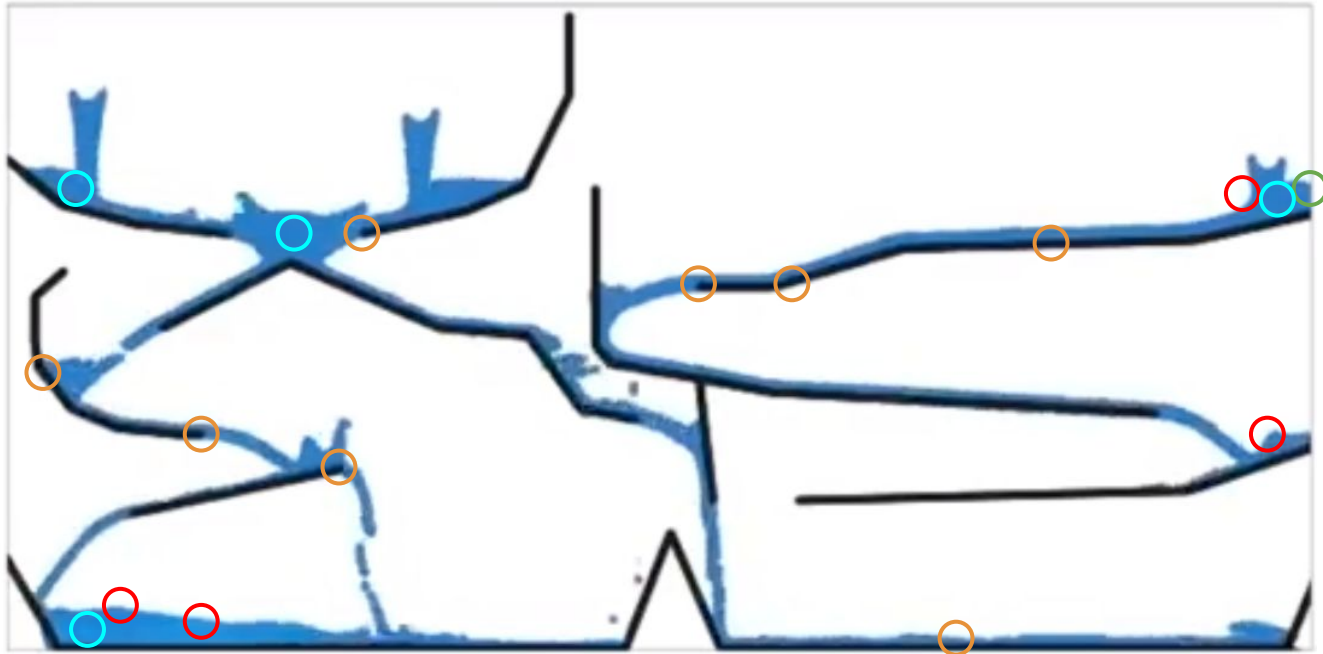
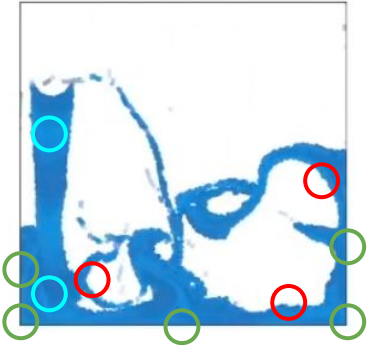
Interactions with obstacles

Water surface dynamics

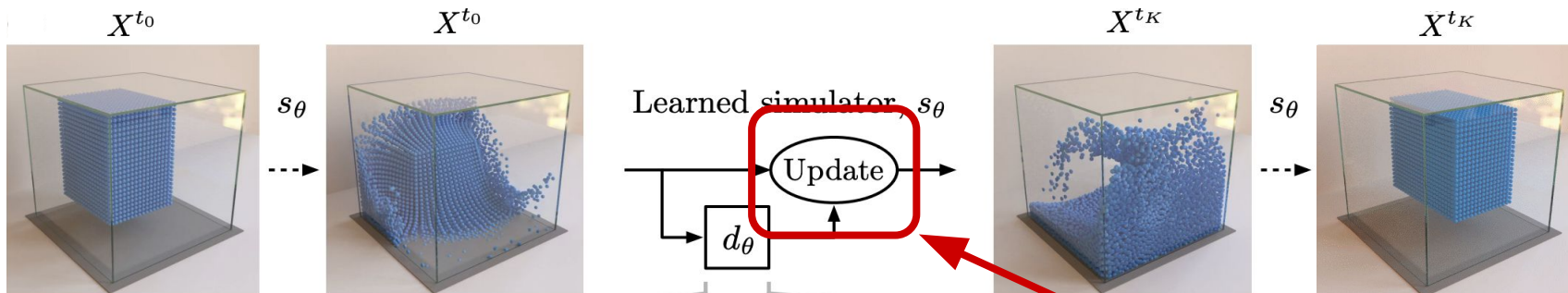
Interaction with box boundaries

Dense blocks of water

Generalization



Physics-inspired inductive biases



**Spatial
equivariance**

**Local
interactions**

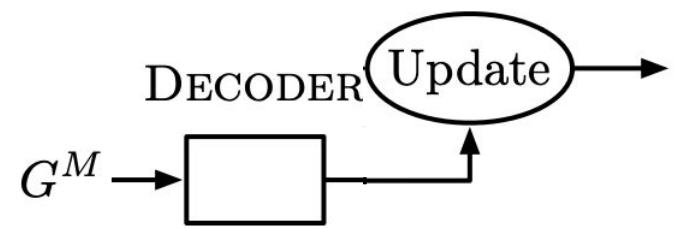
**Universal
rules**

**Pairwise
interactions
Permutation equivariance**

**Superposition
principle**

**Differential
equations**

Decoder and update (Newtonian system)

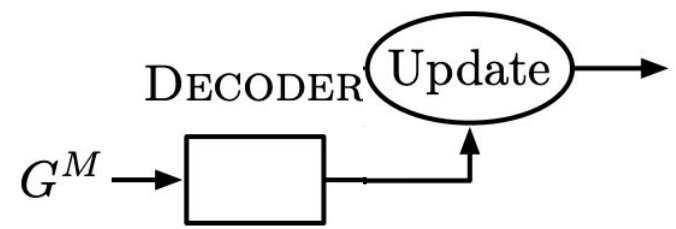


- Naive approach
 - $\text{pos}^{t+1} = \text{NN}(\text{pos}^t, \text{vel}^t)$

Hard to predict static dynamics → Wrong prior



Decoder and update (Newtonian system)



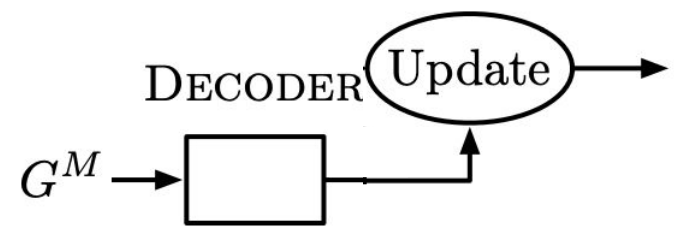
- Less naive approach
 - $\text{pos}^{t+1} = \text{pos}^t + \text{NN}(\text{pos}^t, \text{vel}^t)$

Easy to predict static dynamics

Hard to predict inertial dynamics → Wrong prior



Decoder and update (Newtonian system)

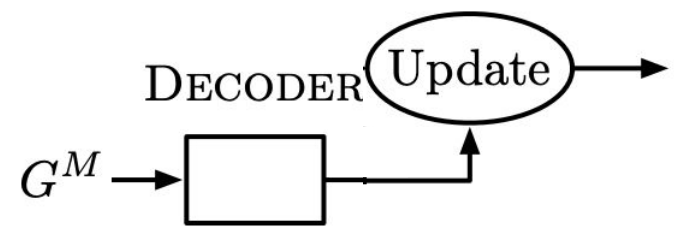


- Better approach
 - $vel^{t+1} = vel^t + NN(pos^t, vel^t)$
 - $pos^{t+1} = pos^t + vel^{t+1}$

Easy to predict static and inertial dynamics!



Decoder and update



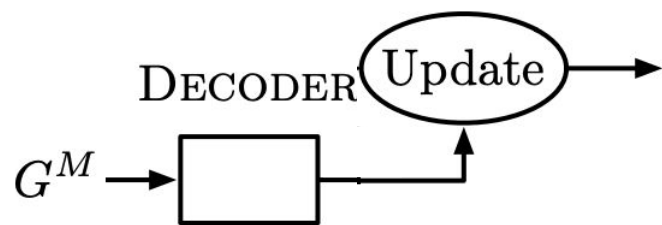
- $\text{vel}^{t+1} = \text{vel}^t + \text{NN}(\text{pos}^t, \text{vel}^t)$
 - $\text{pos}^{t+1} = \text{pos}^t + \text{vel}^{t+1}$
- ← Predicts
"acceleration"

$$\mathbf{F} = m \frac{d\mathbf{v}}{dt} = m\mathbf{a}$$

$$\sum \mathbf{F} = 0 \Leftrightarrow \frac{d\mathbf{v}}{dt} = 0$$



Decoder and update



- $vel^{t+1} = vel^t + NN(pos^t, vel^t) \cdot dt$
- $pos^{t+1} = pos^t + vel^{t+1} \cdot dt$

Euler integrator!

**Differential
equations
(Newton)**

- Reformulate as an ODE:
 - Choose integrator

$$\frac{d \text{ vel}}{dt} = f(\text{pos}, \text{vel})$$

$$\frac{d \text{ pos}}{dt} = \text{vel}$$



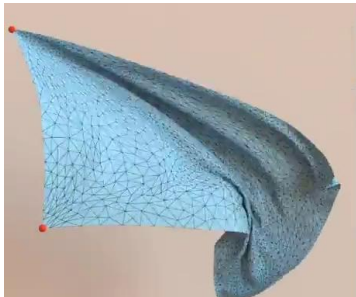
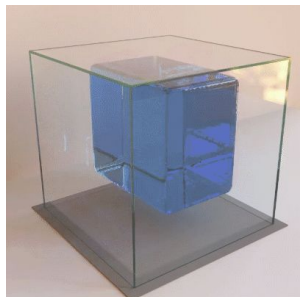
Decoder and update

Newtonian

Particle-based fluids

Cloth simulation

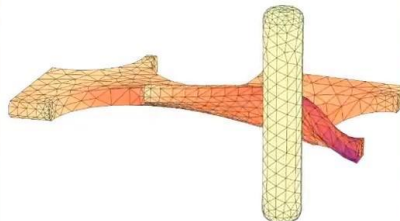
$$\frac{d \dot{\mathbf{x}}}{dt} = f(\mathbf{x}, \dot{\mathbf{x}})$$



Quasistatic

Structural mechanics

$$\frac{d \mathbf{x}}{dt} = f(\mathbf{x})$$



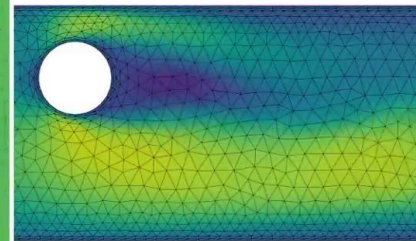
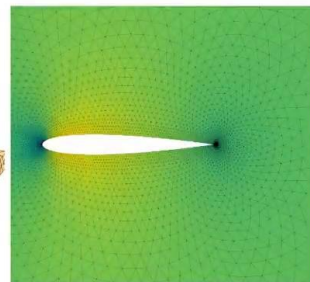
Eulerian Navier Stokes

Aerodynamics

Incompressible fluid

$$\begin{aligned} \frac{d \dot{\mathbf{v}}}{dt} &= f(\mathbf{v}, \rho) \\ \frac{d \dot{\rho}}{dt} &= f(\mathbf{v}, \rho) \end{aligned}$$

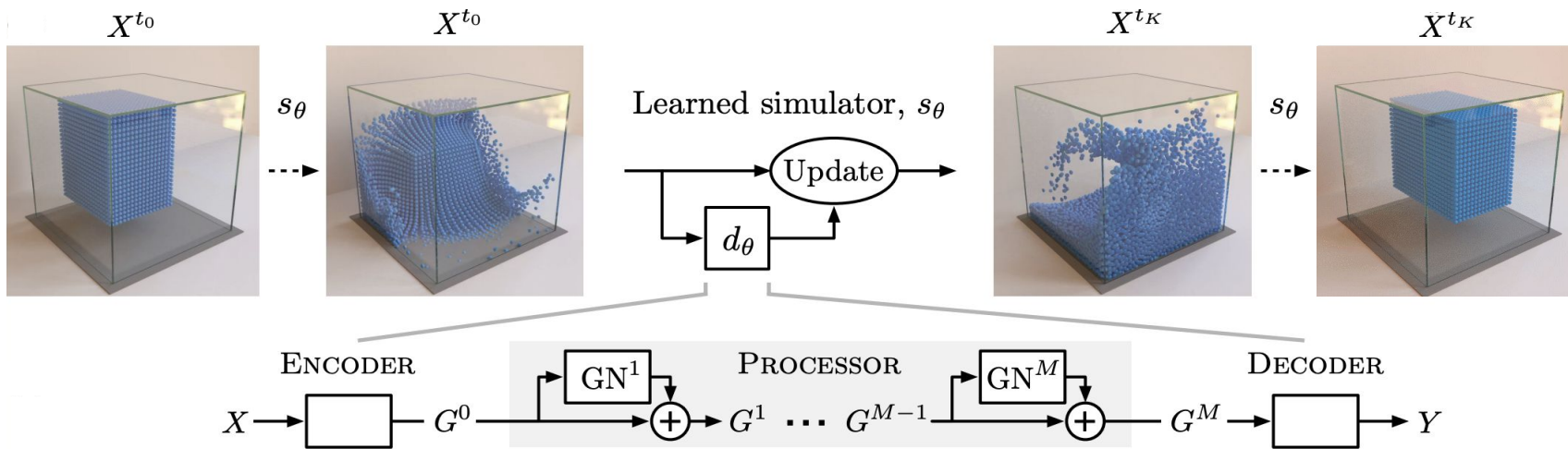
$$\frac{d \dot{\mathbf{v}}}{dt} = f(\mathbf{v})$$



Differential equations



Training tricks



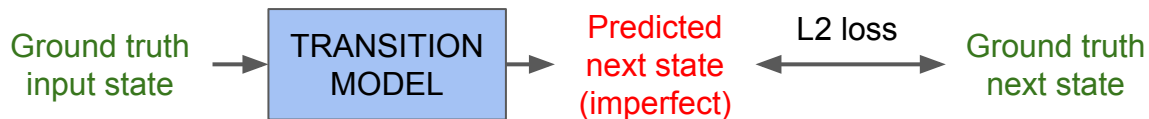
**Training
noise**

**Optimization
tricks**



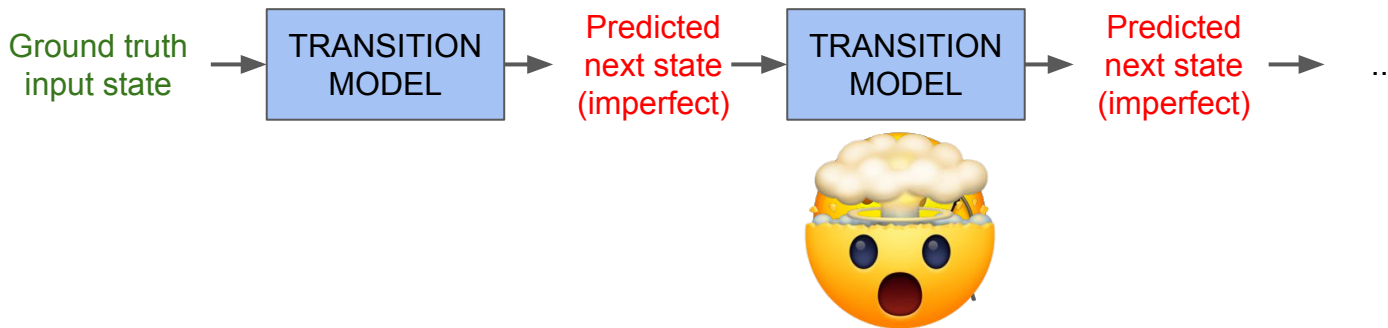
Generalizing to more training steps: training with noise

Train time (one step)



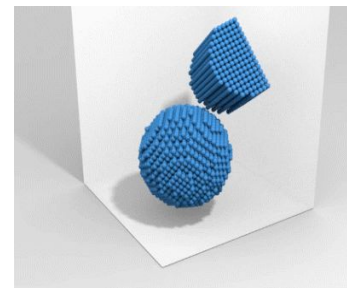
Test time

- Model has never seen imperfect inputs



video credit:

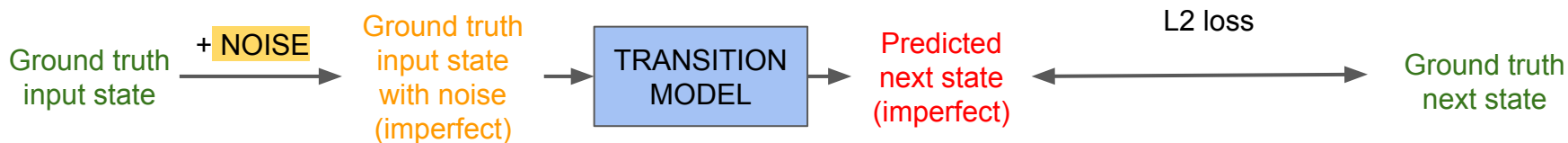
Ummenhofer et al. 2020,
Lagrangian Fluid Simulation with
Continuous Convolutions



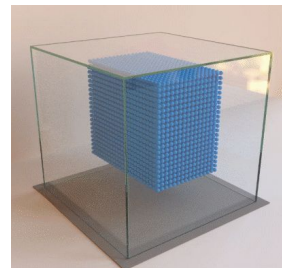
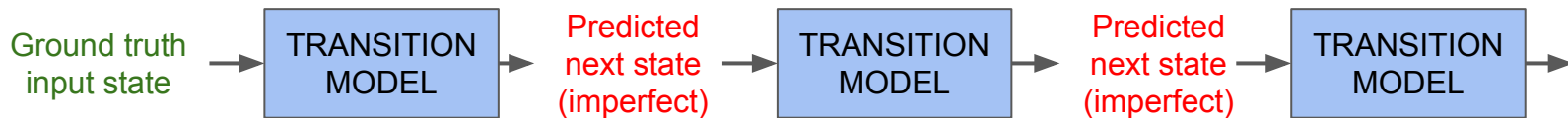
Generalizing to more training steps: training with noise

Train time (with noise)

- Models learns to compensate imperfect inputs during training



Test time (error stays bounded)



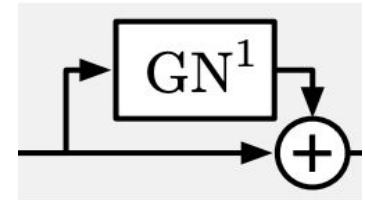
Optimization tricks

- Normalization of representations:
 - Weight initialization: $\text{MLP}(\sim N(0, 1)) \rightarrow \sim N(0, 1)$
 - Normalization of MLP inputs: $\mu=0, \sigma=1$
 - Normalization of MLP outputs: $\mu=0, \sigma=1$
 - LayerNorm: $\mu=0, \sigma=1$ everywhere else!



Optimization tricks

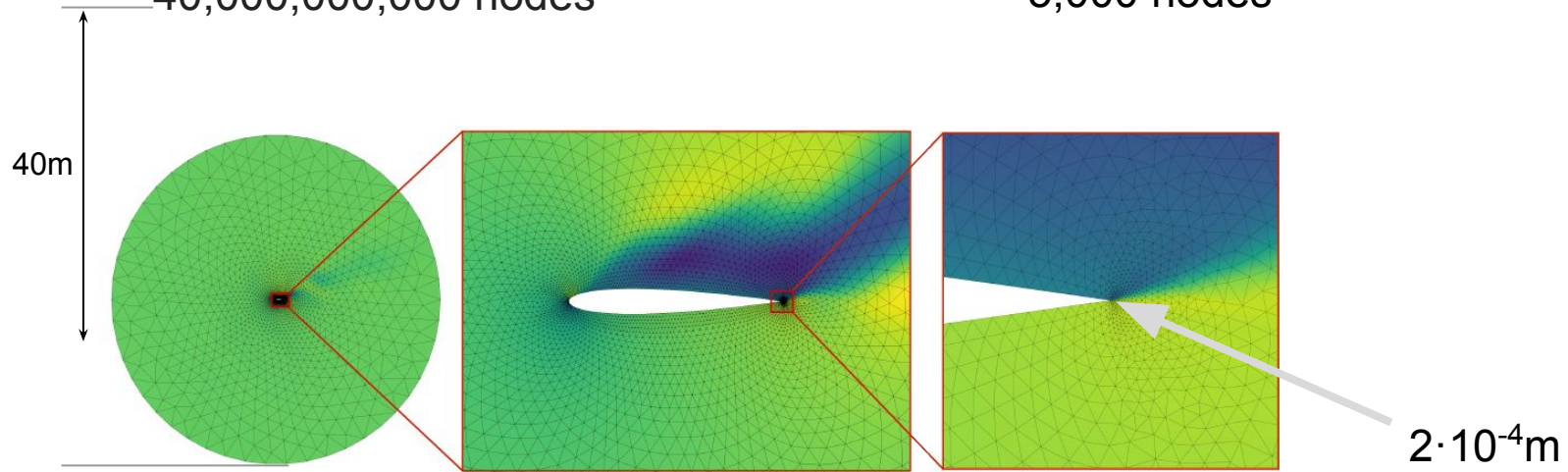
- Normalization of representations:
 - Weight initialization: $\text{MLP}(\sim N(0, 1)) \rightarrow \sim N(0, 1)$
 - Normalization of MLP inputs: $\mu=0, \sigma=1$
 - Normalization of MLP outputs: $\mu=0, \sigma=1$
 - LayerNorm: $\mu=0, \sigma=1$ everywhere else!
- Residual connections
 - At every message passing step



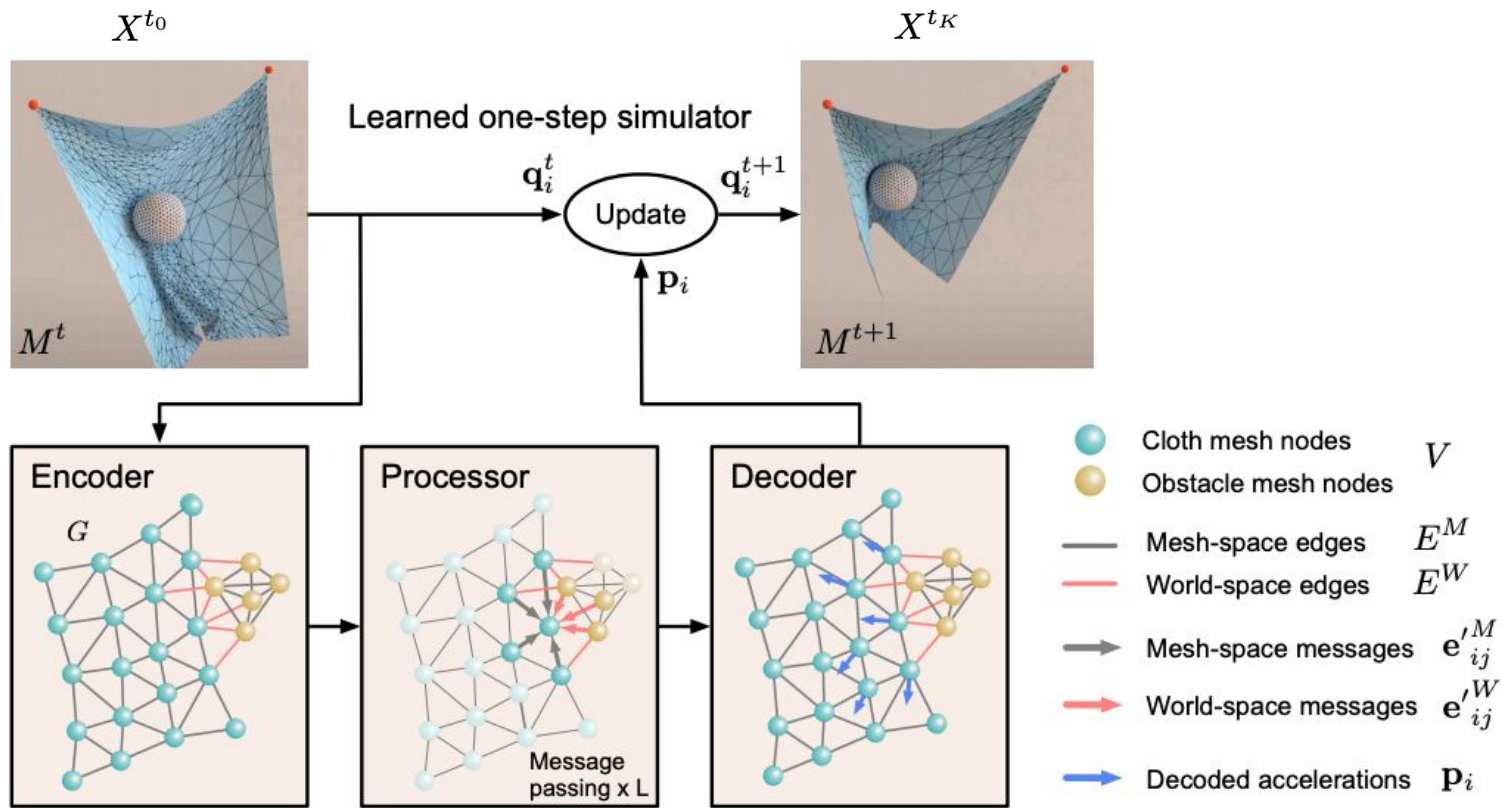
MeshGraphNets: Extension to meshes

uniform grid (e.g.convnet) at $2 \cdot 10^{-4}m$:
40,000,000,000 nodes

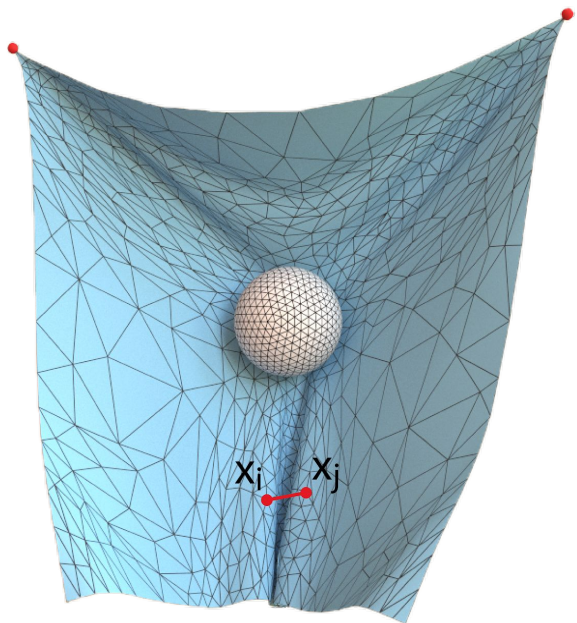
adaptive mesh:
5,000 nodes



MeshGraphNets: Extension to meshes

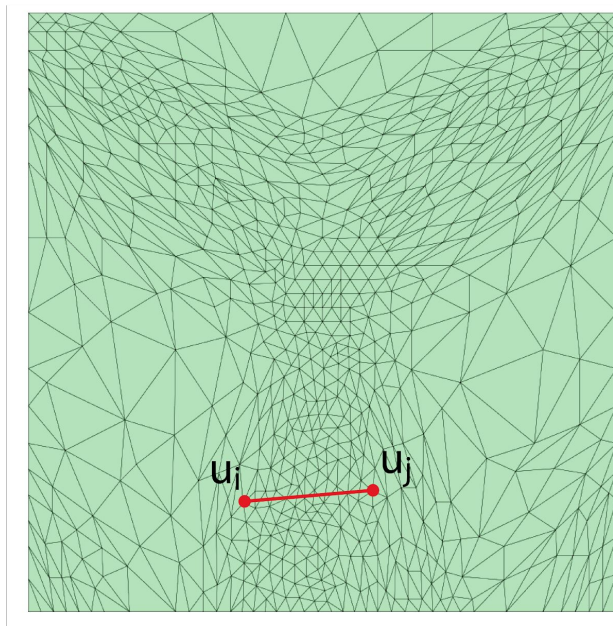


Dual space message passing



world space \mathbf{x}

external dynamics:
computing e.g. collision and contact

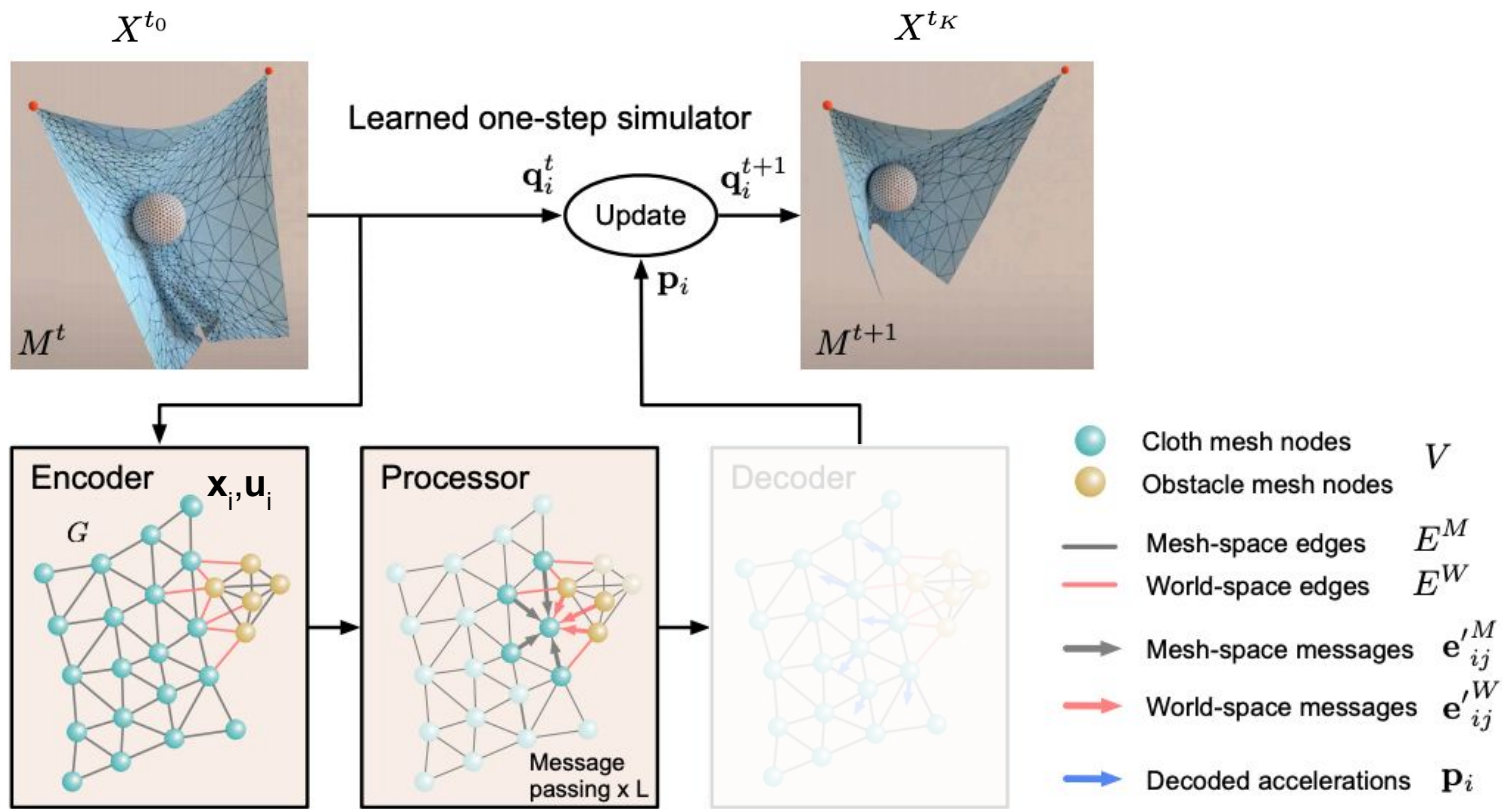


mesh space \mathbf{u}

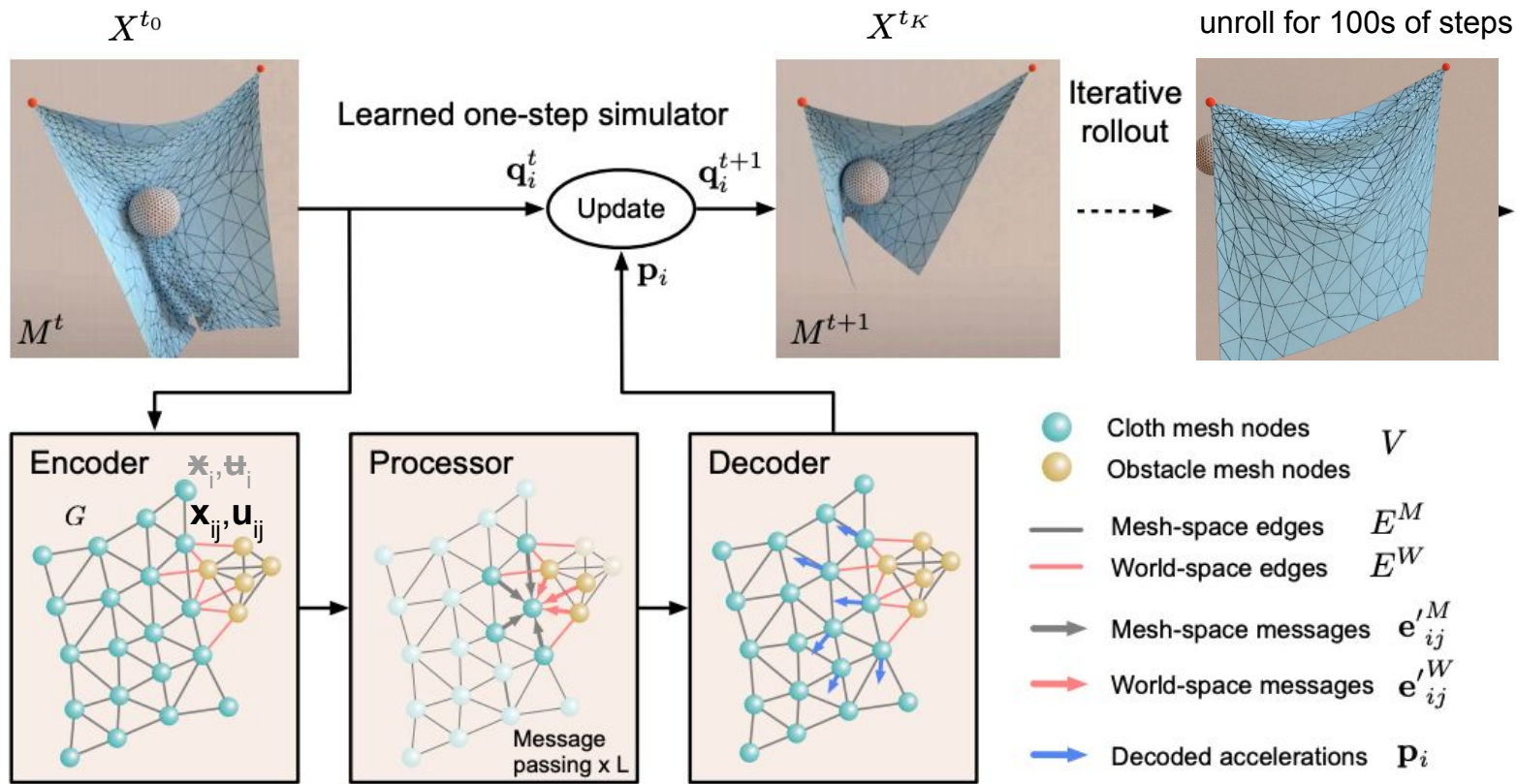
internal dynamics:
estimating differential operators \rightarrow e.g. $\mathbf{F} = \delta \mathbf{x} / \delta \mathbf{u}$
on the simulation manifold



MeshGraphNets: Extension to meshes

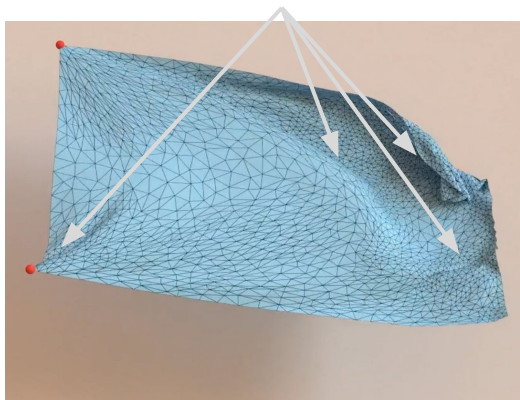


MeshGraphNets: Extension to meshes

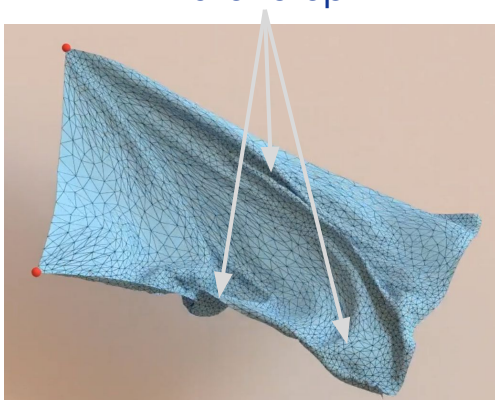


Learned adaptive remeshing

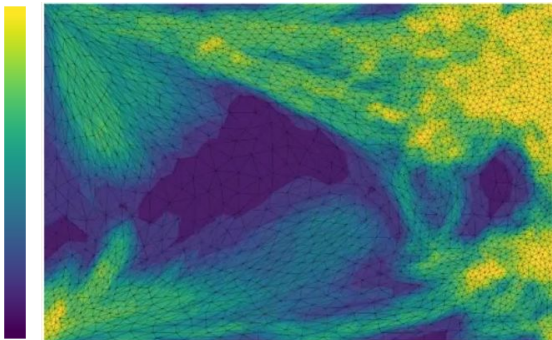
Fine-scale dynamics regions



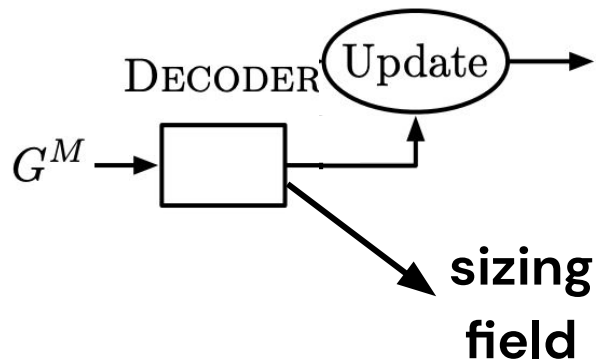
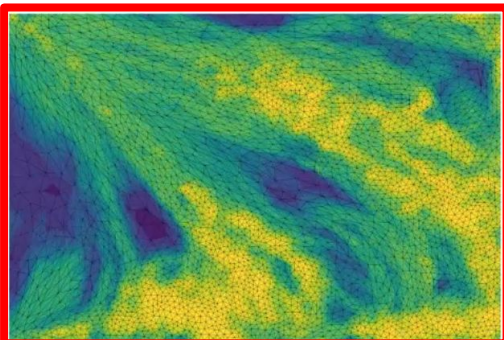
Fine-scale dynamics regions at later step



Sizing field in mesh space



Sizing field in mesh space

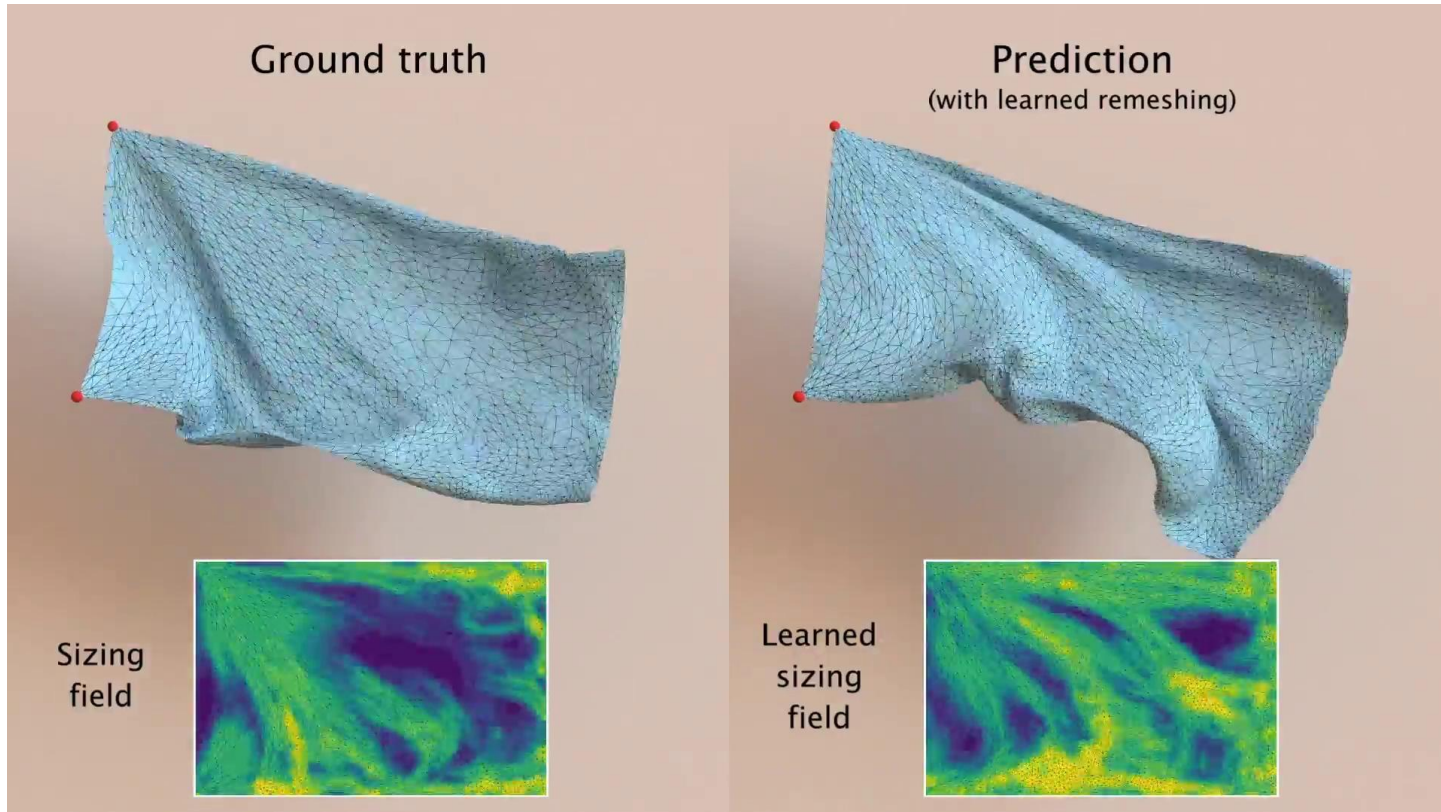


**Predict sizing field
and remesh!**

Resolution



Learned adaptive remeshing



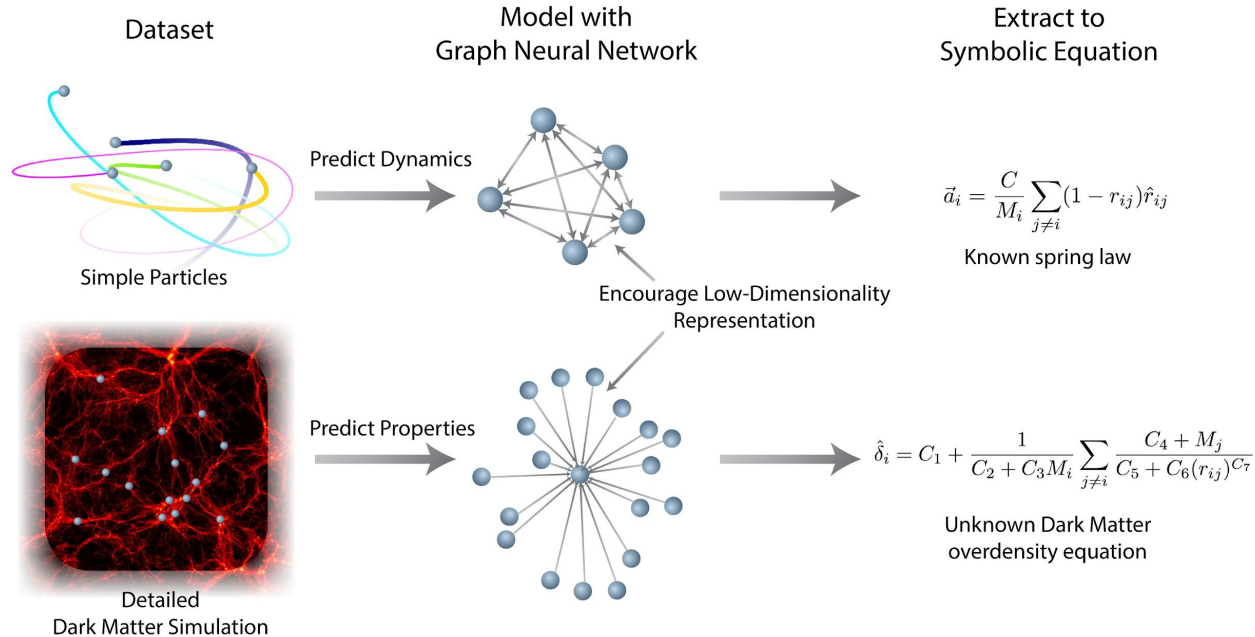
Bonus slide: additional inductive biases

- Hamiltonian inductive bias with Graph Networks
 - “Hamiltonian Graph Networks with ODE Integrators”
Sanchez-Gonzalez et al., 2019, arXiv/NeurIPS 2019 workshop
 - Makes use of the full Graph Network (including global update).
 - Explores additional integrators
- Lagrangian inductive bias with Graph Networks
 - “Lagrangian Neural Networks”
Cranmer et al., 2020, arXiv/ICLR 2020 workshop



Bonus slide: interpretable graph networks

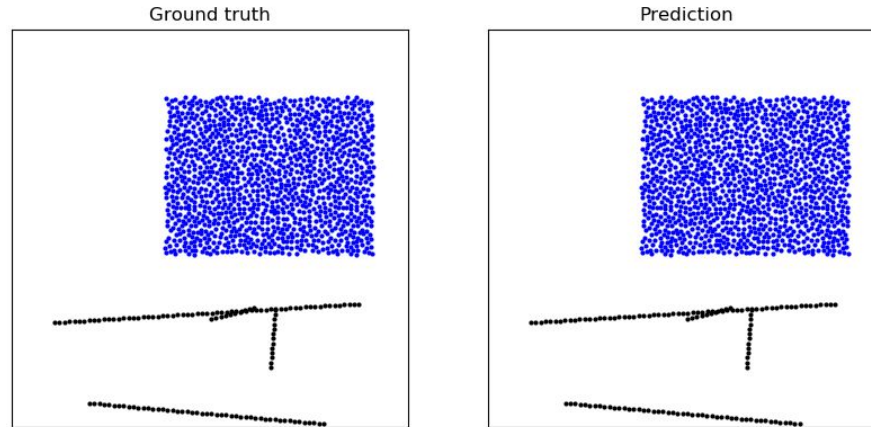
- “Discovering Symbolic Models from Deep Learning with Inductive Biases”
Cranmer et al., NeurIPS 2020
- Extract symbolic models from edge and node functions of a GraphNet



Datasets and source code

- “Learning to Simulate Complex Physics with Graph Networks”

- github.com/deepmind/deepmind-research/tree/master/learning_to_simulate



- Tensorflow 1 & Sonnet 1 (GPU)
- DeepMind Graph Nets library (TF1 and TF2 compatible)
 - github.com/deepmind/graph_nets



learned_simulator.py (_build)

- One step model interface:

```
def _build(self, position_sequence, n_particles_per_example,
           global_context=None, particle_types=None):

    input_graphs_tuple = self._encoder_preprocessor(
        position_sequence, n_particles_per_example, global_context,
        particle_types)

    normalized_acceleration = self._graph_network(input_graphs_tuple)

    next_position = self._decoder_postprocessor(
        normalized_acceleration, position_sequence)

    return next_position
```



learned_simulator.py (_build)

- One step model interface:

```
def _build(self, position_sequence, n_particles_per_example,
           global_context=None, particle_types=None):

    input_graphs_tuple = self._encoder_preprocessor(
        position_sequence, n_particles_per_example, global_context,
        particle_types)

    normalized_acceleration = self._graph_network(input_graphs_tuple)

    next_position = self._decoder_postprocessor(
        normalized_acceleration, position_sequence)

    return next_position
```



learned_simulator.py (_encoder_preprocessor)

- Fixed radius connectivity:

```
# Extract important features from the position_sequence.
most_recent_position = position_sequence[:, -1]

# Get connectivity of the graph.
(senders, receivers, n_edge
 ) = connectivity_utils.compute_connectivity_for_batch_pyfunc(
     most_recent_position, n_node, self._connectivity_radius)
```



learned_simulator.py (_encoder_preprocessor)

- Normalized node features:

```
# Normalized velocity sequence, merging spatial an time axis.
velocity_sequence = time_diff(position_sequence) # Finite-difference.
velocity_stats = self._normalization_stats["velocity"]
normalized_velocity_sequence = (
    velocity_sequence - velocity_stats.mean) / velocity_stats.std

flat_velocity_sequence = snt.MergeDims(start=1, size=2)(
    normalized_velocity_sequence)
node_features.append(flat_velocity_sequence)

# Particle type.
if self._num_particle_types > 1:
    particle_type_embeddings = tf.nn.embedding_lookup(
        self._particle_type_embedding, particle_types)
    node_features.append(particle_type_embeddings)
```



learned_simulator.py (_encoder_preprocessor)

- Normalized edge features:

```
# Collect edge features.
edge_features = []

# Relative displacement and distances normalized to radius
normalized_relative_displacements = (
    tf.gather(most_recent_position, senders) -
    tf.gather(most_recent_position, receivers)) / self._connectivity_radius

edge_features.append(normalized_relative_displacements)

normalized_relative_distances = tf.norm(
    normalized_relative_displacements, axis=-1, keepdims=True)
edge_features.append(normalized_relative_distances)
```



learned_simulator.py (_encoder_preprocessor)

- Normalized global features:

```
# Normalize the global context.
if global_context is not None:
    context_stats = self._normalization_stats["context"]
    # Context in some datasets are all zero, so add an epsilon for numerical
    # stability.
    global_context = (global_context - context_stats.mean) / tf.math.maximum(
        context_stats.std, STD_EPSILON)
```



learned_simulator.py (_encoder_preprocessor)

- Build a GraphsTuple with node, edge and global features:

```
return gn.graphs.GraphsTuple(  
    nodes=tf.concat(node_features, axis=-1),  
    edges=tf.concat(edge_features, axis=-1),  
    globals=global_context,  
    n_node=n_node, n_edge=n_edge,  
    senders=senders, receivers=receivers,)
```

```
# Copy the globals to all of the nodes, if applicable.  
if input_graph.globals is not None:  
    broadcasted_globals = gn.blocks.broadcast_globals_to_nodes(input_graph)  
    input_graph = input_graph.replace(  
        nodes=tf.concat([input_graph.nodes, broadcasted_globals], axis=-1),  
        globals=None)
```



learned_simulator.py (_build)

- One step model interface:

```
def _build(self, position_sequence, n_particles_per_example,
           global_context=None, particle_types=None):

    input_graphs_tuple = self._encoder_preprocessor(
        position_sequence, n_particles_per_example, global_context,
        particle_types)

    normalized_acceleration = self._graph_network(input_graphs_tuple)

    next_position = self._decoder_postprocessor(
        normalized_acceleration, position_sequence)

    return next_position
```

```
self._graph_network = graph_network.EncodeProcessDecode(
    output_size=num_dimensions, **graph_network_kwargs)
```



graph_network.py (_networks_builder)

- Build off-the-shelf Graph Networks:

```
# The encoder graph network independently encodes edge and node features.
encoder_kwargs = dict(
    edge_model_fn=build_mlp_with_layer_norm,
    node_model_fn=build_mlp_with_layer_norm)
self._encoder_network = gn.modules.GraphIndependent(**encoder_kwargs)

# Create `num_message_passing_steps` Interaction Networks with unshared
# parameters that update the node and edge latent features.
self._processor_networks = []
for _ in range(self._num_message_passing_steps):
    self._processor_networks.append(
        gn.modules.InteractionNetwork(
            edge_model_fn=build_mlp_with_layer_norm,
            node_model_fn=build_mlp_with_layer_norm))

# The decoder MLP decodes node latent features into the output size.
self._decoder_network = build_mlp(
    hidden_size=self._mlp_hidden_size,
    num_hidden_layers=self._mlp_num_hidden_layers,
    output_size=self._output_size)
```



graph_network.py (_build)

- Embed features, run message passing and device node features:

```
def _build(self, input_graph: gn.graphs.GraphsTuple) -> tf.Tensor:
    """Forward pass of the learnable dynamics model."""

    # Encode the input graph.
    latent_graph_0 = self._encoder_network(input_graph)

    # Do `m` message passing steps in the latent graphs.
    latent_graph_m = self._process(latent_graph_0)

    # Decode from the last latent graph.
    return self._decoder_network(latent_graph_m.nodes)
```



graph_network.py (_process)

- Several steps of message passing with residual connections:

```
latent_graph_prev_k = latent_graph_0
# Do `m` message passing steps in the latent graphs.
for processor_network_k in self._processor_networks:
    # One step of message passing.
    latent_graph_k = processor_network_k(latent_graph_prev_k)

    # Add residuals with previous layer.
    latent_graph_k = latent_graph_k.replace(
        nodes=latent_graph_k.nodes+latent_graph_prev_k.nodes,
        edges=latent_graph_k.edges+latent_graph_prev_k.edges)
    latent_graph_prev_k = latent_graph_k

latent_graph_m = latent_graph_k
```



learned_simulator.py (_build)

- One step model interface:

```
def _build(self, position_sequence, n_particles_per_example,
           global_context=None, particle_types=None):

    input_graphs_tuple = self._encoder_preprocessor(
        position_sequence, n_particles_per_example, global_context,
        particle_types)

    normalized_acceleration = self._graph_network(input_graphs_tuple)

    next_position = self._decoder_postprocessor(
        normalized_acceleration, position_sequence)

    return next_position
```



learned_simulator.py (_decoder_postprocessor)

- Inverse normalization and Euler Integrator:

```
def _decoder_postprocessor(self, normalized_acceleration, position_sequence):  
  
    # The model produces the output in normalized space so we apply inverse  
    # normalization.  
    acceleration_stats = self._normalization_stats["acceleration"]  
    acceleration = (  
        normalized_acceleration * acceleration_stats.std  
    ) + acceleration_stats.mean  
  
    # Use an Euler integrator to go from acceleration to position, assuming  
    # a dt=1 corresponding to the size of the finite difference.  
    most_recent_position = position_sequence[:, -1]  
    most_recent_velocity = most_recent_position - position_sequence[:, -2]  
  
    new_velocity = most_recent_velocity + acceleration # * dt = 1  
    new_position = most_recent_position + new_velocity # * dt = 1  
    return new_position
```



Conclusions

- **Graph Networks are powerful models** for everyone!
- Architectures for simulation (**mesh-based and-particle based**)
- **General principles** to find well-matched **inductive biases**
- **Effective model training tricks**
- A **reference implementation** for one of these models at https://github.com/deepmind/deepmind-research/tree/master/learning_to_simulate



DeepMind

Deep Dive on Graph Networks for Learning Simulation

Thanks for your attention!
Question time?

Inter-experiment Machine Learning Workshop
22 Oct 2020

Alvaro Sanchez-Gonzalez - DeepMind

Victor Bapst, Peter Battaglia, Kyle Cranmer, Miles Cranmer,
Meino Fortunato, Jonathan Godwin, Jessica Hamrick,
Shirley Ho, Jure Leskovec, Tobias Pfaff, Rex Ying,