**Simulating Complex Physics with Graph Networks: step by step**

**Overview**

• By Peng Chen, Shiyu Li, Haochen Shi as part of Stanford CS224W course project.

• This tutorial provides a step-by-step guide for how to build a Graph Network to simulate complex physics.

**Before we get started:**

• This Colab includes a concise PyG implementation of paper \*\*\*Learning to Simulate Complex Physics with Graph Networks\*.

• We adapted code from open-source tensorflow implementation by DeepMind.

* + Link to pdf of this paper: <https://arxiv.org/abs/2002.09405>
  + Link to Deepmind's implementation: <https://github.com/deepmind/deepmind-research/tree/master/learning_to_simulate>
  + Link to video site by DeepMind: <https://sites.google.com/view/learning-to-simulate>
* Make sure to **sequentially run all cells in each section**, so that intermediate variables / packages will carry over to next cell.
* Feel free to make a copy to your own drive to play around with it! Have fun with this tutorial!

**Device**

• We recommend using a GPU for this Colab.

• click Runtime and then Change runtime type.

• Then set hardware accelerator to GPU.

**Setup**

• installation of PyG on Colab may be bit tricky.

• Before we get started, let's check which version of PyTorch you are running.

In [ ]:

**!**pip install torch

In [ ]:

**!**conda install pytorch-cpu torchvision-cpu -c pytorch

In [4]:

**import** os

**import** torch

print(f"PyTorch has version {torch**.**\_\_version\_\_} with cuda {torch**.**version**.**cuda}")

Download necessary packages for PyG. Make sure that your version of torch matches output from cell above. In case of any issues, more information can be found on [PyG's installation page](https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html)

<https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html>

In [2]:

*# Install torch geometric*

**!**pip install torch-cluster -f https://data.pyg.org/whl/torch-1.10.0+cu111.html

**!**pip install torch-scatter -f https://data.pyg.org/whl/torch-1.10.0+cu111.html

**!**pip install torch-sparse -f https://data.pyg.org/whl/torch-1.10.0+cu111.html

**!**pip install torch-geometric

**Dataset**

• dataset WaterDropSmall includes 100 videos of dropping water to ground rendered in a particle-based physics simulator.

• It is a cropped version of WaterDrop dataset by Deepmind.

• we download this dataset from Google Cloud stoarge to folder temp/datasets in file system.

• may inspect downloaded files on Files menu on left of this Colab.

metadata.json file in dataset includes following information:

1. sequence length of each video data point
2. dimensionality, 2d or 3d
3. box bounds, which specify bounding box for scene
4. default connectivity radius, which defines size of each particle's neighborhood
5. statistics for normalization, such as mean and standard deviation of velocity and acceleration of particles

Each data point in dataset includes following information:

1. type of particles, such as water
2. particle positions at each frame in video

In [1]:

DATASET\_NAME **=** "WaterDropSmall"

OUTPUT\_DIR **=** os**.**path**.**join("temp", "datasets", DATASET\_NAME)

BASE\_URL **=** f"https://storage.googleapis.com/cs224w\_course\_project\_dataset/{DATASET\_NAME}"

**!**mkdir -p "$OUTPUT\_DIR"

META\_DATA\_PATH **=** f"{OUTPUT\_DIR}/metadata.json"

CLOUD\_PATH **=** f"{BASE\_URL}/metadata.json"

**!**wget -O "$META\_DATA\_PATH" "$CLOUD\_PATH"

**for** split **in** ["test", "train", "valid"]:

**for** suffix **in** ["offset.json", "particle\_type.dat", "position.dat"]:

DATA\_PATH **=** f"{OUTPUT\_DIR}/{split}\_{suffix}"

CLOUD\_PATH **=** f"{BASE\_URL}/{split}\_{suffix}"

**!**wget -O "$DATA\_PATH" "$CLOUD\_PATH"

## Data Preprocessing

Since we cannot apply raw data in dataset to train GNN model directly, we need to go through following steps to convert raw data into graphs with descriptive node features and edge features:

1. Apply noise to trajectory to have more diverse training examples
2. Construct graph based on distance between particles
3. Extract node-level features: particle velocities and their distance to boundary
4. Extract edge-level features: displacement and distance between particles

If you are not interested in data pipeline, your can skip to end of this section. There is a detailed explanation and visualization of one data point.

In [ ]:

**import** json

**import** numpy **as** np

**import** torch\_geometric **as** pyg

**def** generate\_noise(position\_seq, noise\_std):

"""Generate noise for a trajectory"""

velocity\_seq **=** position\_seq[:, 1:] **-** position\_seq[:, :**-**1]

time\_steps **=** velocity\_seq**.**size(1)

velocity\_noise **=** torch**.**randn\_like(velocity\_seq) **\*** (noise\_std **/** time\_steps **\*\*** 0.5)

velocity\_noise **=** velocity\_noise**.**cumsum(dim**=**1)

position\_noise **=** velocity\_noise**.**cumsum(dim**=**1)

position\_noise **=** torch**.**cat((torch**.**zeros\_like(position\_noise)[:, 0:1], position\_noise), dim**=**1)

**return** position\_noise

**def** preprocess(particle\_type, position\_seq, target\_position, metadata, noise\_std):

"""Preprocess a trajectory and construct graph"""

*# apply noise to trajectory*

position\_noise **=** generate\_noise(position\_seq, noise\_std)

position\_seq **=** position\_seq **+** position\_noise

*# calculate velocities of particles*

recent\_position **=** position\_seq[:, **-**1]

velocity\_seq **=** position\_seq[:, 1:] **-** position\_seq[:, :**-**1]

*# construct graph based on distances between particles*

n\_particle **=** recent\_position**.**size(0)

edge\_index **=** pyg**.**nn**.**radius\_graph(recent\_position, metadata["default\_connectivity\_radius"], loop**=True**, max\_num\_neighbors**=**n\_particle)

*# node-level features: velocity, distance to boundary*

normal\_velocity\_seq **=** (velocity\_seq **-** torch**.**tensor(metadata["vel\_mean"])) **/** torch**.**sqrt(torch**.**tensor(metadata["vel\_std"]) **\*\*** 2 **+** noise\_std **\*\*** 2)

boundary **=** torch**.**tensor(metadata["bounds"])

distance\_to\_lower\_boundary **=** recent\_position **-** boundary[:, 0]

distance\_to\_upper\_boundary **=** boundary[:, 1] **-** recent\_position

distance\_to\_boundary **=** torch**.**cat((distance\_to\_lower\_boundary, distance\_to\_upper\_boundary), dim**=-**1)

distance\_to\_boundary **=** torch**.**clip(distance\_to\_boundary **/** metadata["default\_connectivity\_radius"], **-**1.0, 1.0)

*# edge-level features: displacement, distance*

dim **=** recent\_position**.**size(**-**1)

edge\_displacement **=** (torch**.**gather(recent\_position, dim**=**0, index**=**edge\_index[0]**.**unsqueeze(**-**1)**.**expand(**-**1, dim)) **-**

torch**.**gather(recent\_position, dim**=**0, index**=**edge\_index[1]**.**unsqueeze(**-**1)**.**expand(**-**1, dim)))

edge\_displacement **/=** metadata["default\_connectivity\_radius"]

edge\_distance **=** torch**.**norm(edge\_displacement, dim**=-**1, keepdim**=True**)

*# ground truth for training*

**if** target\_position **is** **not** **None**:

last\_velocity **=** velocity\_seq[:, **-**1]

next\_velocity **=** target\_position **+** position\_noise[:, **-**1] **-** recent\_position

acceleration **=** next\_velocity **-** last\_velocity

acceleration **=** (acceleration **-** torch**.**tensor(metadata["acc\_mean"])) **/** torch**.**sqrt(torch**.**tensor(metadata["acc\_std"]) **\*\*** 2 **+** noise\_std **\*\*** 2)

**else**:

acceleration **=** **None**

*# return graph with features*

graph **=** pyg**.**data**.**Data(

x**=**particle\_type,

edge\_index**=**edge\_index,

edge\_attr**=**torch**.**cat((edge\_displacement, edge\_distance), dim**=-**1),

y**=**acceleration,

pos**=**torch**.**cat((velocity\_seq**.**reshape(velocity\_seq**.**size(0), **-**1), distance\_to\_boundary), dim**=-**1)

)

**return** graph

### One Step Dataset

• Each datapoint in this dataset contains trajectories sliced to short time windows.

• we use this dataset in training phase because history of particles' states are necessary for model to make predictions.

• But in meantime, since long-horizon prediction is inaccurate and time-consuming, we sliced trajectories to short time windows to improve perfomance of model.

In [ ]:

**class** OneStepDataset(pyg**.**data**.**Dataset):

**def** \_\_init\_\_(self, data\_path, split, window\_length**=**7, noise\_std**=**0.0, return\_pos**=False**):

super()**.**\_\_init\_\_()

*# load dataset from disk*

**with** open(os**.**path**.**join(data\_path, "metadata.json")) **as** f:

self**.**metadata **=** json**.**load(f)

**with** open(os**.**path**.**join(data\_path, f"{split}\_offset.json")) **as** f:

self**.**offset **=** json**.**load(f)

self**.**offset **=** {int(k): v **for** k, v **in** self**.**offset**.**items()}

self**.**window\_length **=** window\_length

self**.**noise\_std **=** noise\_std

self**.**return\_pos **=** return\_pos

self**.**particle\_type **=** np**.**memmap(os**.**path**.**join(data\_path, f"{split}\_particle\_type.dat"), dtype**=**np**.**int64, mode**=**"r")

self**.**position **=** np**.**memmap(os**.**path**.**join(data\_path, f"{split}\_position.dat"), dtype**=**np**.**float32, mode**=**"r")

**for** traj **in** self**.**offset**.**values():

self**.**dim **=** traj["position"]["shape"][2]

**break**

*# cut particle trajectories according to time slices*

self**.**windows **=** []

**for** traj **in** self**.**offset**.**values():

size **=** traj["position"]["shape"][1]

length **=** traj["position"]["shape"][0] **-** window\_length **+** 1

**for** i **in** range(length):

desc **=** {

"size": size,

"type": traj["particle\_type"]["offset"],

"pos": traj["position"]["offset"] **+** i **\*** size **\*** self**.**dim,

}

self**.**windows**.**append(desc)

**def** len(self):

**return** len(self**.**windows)

**def** get(self, idx):

*# load corresponding data for this time slice*

window **=** self**.**windows[idx]

size **=** window["size"]

particle\_type **=** self**.**particle\_type[window["type"]: window["type"] **+** size]**.**copy()

particle\_type **=** torch**.**from\_numpy(particle\_type)

position\_seq **=** self**.**position[window["pos"]: window["pos"] **+** self**.**window\_length **\*** size **\*** self**.**dim]**.**copy()

position\_seq**.**resize(self**.**window\_length, size, self**.**dim)

position\_seq **=** position\_seq**.**transpose(1, 0, 2)

target\_position **=** position\_seq[:, **-**1]

position\_seq **=** position\_seq[:, :**-**1]

target\_position **=** torch**.**from\_numpy(target\_position)

position\_seq **=** torch**.**from\_numpy(position\_seq)

*# construct graph*

**with** torch**.**no\_grad():

graph **=** preprocess(particle\_type, position\_seq, target\_position, self**.**metadata, self**.**noise\_std)

**if** self**.**return\_pos:

**return** graph, position\_seq[:, **-**1]

**return** graph

### Rollout Dataset

• Each datapoint in this dataset contains trajectories of particles over 1000 time frames.

• This dataset used in evaluation phase to measure model's ability to makie long-horizon predictions.

In [ ]:

**class** RolloutDataset(pyg**.**data**.**Dataset):

**def** \_\_init\_\_(self, data\_path, split, window\_length**=**7):

super()**.**\_\_init\_\_()

*# load data from disk*

**with** open(os**.**path**.**join(data\_path, "metadata.json")) **as** f:

self**.**metadata **=** json**.**load(f)

**with** open(os**.**path**.**join(data\_path, f"{split}\_offset.json")) **as** f:

self**.**offset **=** json**.**load(f)

self**.**offset **=** {int(k): v **for** k, v **in** self**.**offset**.**items()}

self**.**window\_length **=** window\_length

self**.**particle\_type **=** np**.**memmap(os**.**path**.**join(data\_path, f"{split}\_particle\_type.dat"), dtype**=**np**.**int64, mode**=**"r")

self**.**position **=** np**.**memmap(os**.**path**.**join(data\_path, f"{split}\_position.dat"), dtype**=**np**.**float32, mode**=**"r")

**for** traj **in** self**.**offset**.**values():

self**.**dim **=** traj["position"]["shape"][2]

**break**

**def** len(self):

**return** len(self**.**offset)

**def** get(self, idx):

traj **=** self**.**offset[idx]

size **=** traj["position"]["shape"][1]

time\_step **=** traj["position"]["shape"][0]

particle\_type **=** self**.**particle\_type[traj["particle\_type"]["offset"]: traj["particle\_type"]["offset"] **+** size]**.**copy()

particle\_type **=** torch**.**from\_numpy(particle\_type)

position **=** self**.**position[traj["position"]["offset"]: traj["position"]["offset"] **+** time\_step **\*** size **\*** self**.**dim]**.**copy()

position**.**resize(traj["position"]["shape"])

position **=** torch**.**from\_numpy(position)

data **=** {"particle\_type": particle\_type, "position": position}

**return** data

### Visualize a graph in dataset

Each data point in dataset is a pyg.data.Data object which describes a graph. We explain contents of first data point, and visualize graph.

In [ ]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

**import** networkx **as** nx

dataset\_sample **=** OneStepDataset(OUTPUT\_DIR, "valid", return\_pos**=True**)

graph, position **=** dataset\_sample[0]

print(f"first item in valid set is a graph: {graph}")

print(f"This graph has {graph**.**num\_nodes} nodes and {graph**.**num\_edges} edges.")

print(f"Each node is a particle and each edge is interaction between two particles.")

print(f"Each node has {graph**.**num\_node\_features} categorial feature (Data.x), which represents type of node.")

print(f"Each node has a {graph**.**pos**.**size(1)}-dim feature vector (Data.pos), which represents positions and velocities of particle (node) in several frames.")

print(f"Each edge has a {graph**.**num\_edge\_features}-dim feature vector (Data.edge\_attr), which represents relative distance and displacement between particles.")

print(f"model is expected to predict a {graph**.**y**.**size(1)}-dim vector for each node (Data.y), which represents acceleration of particle.")

*# remove directions of edges, because it is a symmetric directed graph.*

nx\_graph **=** pyg**.**utils**.**to\_networkx(graph)**.**to\_undirected()

*# remove self loops, because every node has a self loop.*

nx\_graph**.**remove\_edges\_from(nx**.**selfloop\_edges(nx\_graph))

plt**.**figure(figsize**=**(7, 7))

nx**.**draw(nx\_graph, pos**=**{i: tuple(v) **for** i, v **in** enumerate(position)}, node\_size**=**50)

plt**.**show()

first item in valid set is a graph: Data(x=[482], edge\_index=[2, 3070], edge\_attr=[3070, 3], y=[482, 2], pos=[482, 14])

This graph has 482 nodes and 3070 edges.

Each node is a particle and each edge is interaction between two particles.

Each node has 1 categorial feature (Data.x), which represents type of node.

Each node has a 14-dim feature vector (Data.pos), which represents positions and velocities of particle (node) in several frames.

Each edge has a 3-dim feature vector (Data.edge\_attr), which represents relative distance and displacement between particles.

model is expected to predict a 2-dim vector for each node (Data.y), which represents acceleration of particle.

## GNN Model

We will walk through implementation of GNN model in this section!

### Helper class

We first define a class for Multi-Layer Perceptron (MLP). This class generates an MLP given width and depth of it. Because MLPs are used in several places of GNN, this helper class will make code cleaner.

In [ ]:

**import** math

**import** torch\_scatter

**class** MLP(torch**.**nn**.**Module):

"""Multi-Layer perceptron"""

**def** \_\_init\_\_(self, input\_size, hidden\_size, output\_size, layers, layernorm**=True**):

super()**.**\_\_init\_\_()

self**.**layers **=** torch**.**nn**.**ModuleList()

**for** i **in** range(layers):

self**.**layers**.**append(torch**.**nn**.**Linear(

input\_size **if** i **==** 0 **else** hidden\_size,

output\_size **if** i **==** layers **-** 1 **else** hidden\_size,

))

**if** i **!=** layers **-** 1:

self**.**layers**.**append(torch**.**nn**.**ReLU())

**if** layernorm:

self**.**layers**.**append(torch**.**nn**.**LayerNorm(output\_size))

self**.**reset\_parameters()

**def** reset\_parameters(self):

**for** layer **in** self**.**layers:

**if** isinstance(layer, torch**.**nn**.**Linear):

layer**.**weight**.**data**.**normal\_(0, 1 **/** math**.**sqrt(layer**.**in\_features))

layer**.**bias**.**data**.**fill\_(0)

**def** forward(self, x):

**for** layer **in** self**.**layers:

x **=** layer(x)

**return** x

### GNN layers

In following code block, we implement one type of GNN layer named InteractionNetwork (IN), which is proposed by paper Interaction Networks for Learning about Objects, Relations and Physics.

For a graph G𝐺, let feature of node i𝑖 be vi𝑣𝑖, and feature of edge (i,j)(𝑖,𝑗) be ei,j𝑒𝑖,𝑗. There are three stages for IN to generate new features of nodes and edges.

1. **Message generation.** If there is an edge pointing from node i𝑖 to node j𝑗, node i𝑖 sends a message to node j𝑗. message carries information of edge and its two nodes, so it is generated by following equation Msgi,j=MLP(vi,vj,ei,j)Msg𝑖,𝑗=MLP(𝑣𝑖,𝑣𝑗,𝑒𝑖,𝑗).
2. **Message aggregation.** In this stage, each node of graph aggregates all messages that it received to a fixed-sized representation. In IN, aggregation means summing all messages up, i.e., Aggi=∑(j,i)∈GMsgi,jAgg𝑖=∑(𝑗,𝑖)∈𝐺Msg𝑖,𝑗.
3. **Update.** Finally, we update features of nodes and edges with results of previous stages. For each edge, its new feature is simply sum of its old feature and correspond message, i.e., e′i,j=ei,j+Msgi,j𝑒𝑖,𝑗′=𝑒𝑖,𝑗+Msg𝑖,𝑗. For each node, new feature is determined by its old feature and aggregated message, i.e., v′i=vi+MLP(vi,Aggi)𝑣𝑖′=𝑣𝑖+MLP(𝑣𝑖,Agg𝑖).

In PyG, GNN layers are implemented as subclass of MessagePassing. We need to override three critical functions to implement our InteractionNetwork GNN layer. Each function corresponds to one stage of GNN layer.

1. message() -> message generation

This function controls how a message is generated on each edge of graph. It takes three arguments: (1) x\_i, features of source nodes; (2) x\_j, features of target nodes; and (3) edge\_feature, features of edges themselves. In IN, we simply concatenate all these features and generate messages with an MLP.

1. aggregate() -> message aggregation

This function aggregates messages for nodes. It depends on two arguments: (1) inputs, messages; and (2) index, graph structure. We handle over task of message aggregation to function torch\_scatter.scatter and specifies in argument reduce that we want to sum messages up. Because we want to retain messages themselves to update edge features, we return both messages and aggregated messages.

1. forward() -> update

This function puts everything together. x is node features, edge\_index is graph structure and edge\_feature is edge features. functionMessagePassing.propagate invokes functions message and aggregate for us. Then, we update node features and edge features and return them.

In [ ]:

**class** InteractionNetwork(pyg**.**nn**.**MessagePassing):

"""Interaction Network as proposed in this paper:

https://proceedings.neurips.cc/paper/2016/hash/3147da8ab4a0437c15ef51a5cc7f2dc4-Abstract.html"""

**def** \_\_init\_\_(self, hidden\_size, layers):

super()**.**\_\_init\_\_()

self**.**lin\_edge **=** MLP(hidden\_size **\*** 3, hidden\_size, hidden\_size, layers)

self**.**lin\_node **=** MLP(hidden\_size **\*** 2, hidden\_size, hidden\_size, layers)

**def** forward(self, x, edge\_index, edge\_feature):

edge\_out, aggr **=** self**.**propagate(edge\_index, x**=**(x, x), edge\_feature**=**edge\_feature)

node\_out **=** self**.**lin\_node(torch**.**cat((x, aggr), dim**=-**1))

edge\_out **=** edge\_feature **+** edge\_out

node\_out **=** x **+** node\_out

**return** node\_out, edge\_out

**def** message(self, x\_i, x\_j, edge\_feature):

x **=** torch**.**cat((x\_i, x\_j, edge\_feature), dim**=-**1)

x **=** self**.**lin\_edge(x)

**return** x

**def** aggregate(self, inputs, index, dim\_size**=None**):

out **=** torch\_scatter**.**scatter(inputs, index, dim**=**self**.**node\_dim, dim\_size**=**dim\_size, reduce**=**"sum")

**return** (inputs, out)

### GNN

Now its time to stack GNN layers to a GNN. Besides GNN layers, there are pre-processing and post-processing blocks in GNN. Before GNN layers, input features are transformed by MLP so that expressiveness of GNN is improved without increasing GNN layers. After GNN layers, final outputs (accelerations of particles in our case) are extracted from features generated by GNN layers to meet requirement of task.

In [ ]:

**class** LearnedSimulator(torch**.**nn**.**Module):

"""Graph Network-based Simulators(GNS)"""

**def** \_\_init\_\_(

self,

hidden\_size**=**128,

n\_mp\_layers**=**10, *# number of GNN layers*

num\_particle\_types**=**9,

particle\_type\_dim**=**16, *# embedding dimension of particle types*

dim**=**2, *# dimension of world, typical 2D or 3D*

window\_size**=**5, *# model looks into W frames before frame to be predicted*

):

super()**.**\_\_init\_\_()

self**.**window\_size **=** window\_size

self**.**embed\_type **=** torch**.**nn**.**Embedding(num\_particle\_types, particle\_type\_dim)

self**.**node\_in **=** MLP(particle\_type\_dim **+** dim **\*** (window\_size **+** 2), hidden\_size, hidden\_size, 3)

self**.**edge\_in **=** MLP(dim **+** 1, hidden\_size, hidden\_size, 3)

self**.**node\_out **=** MLP(hidden\_size, hidden\_size, dim, 3, layernorm**=False**)

self**.**n\_mp\_layers **=** n\_mp\_layers

self**.**layers **=** torch**.**nn**.**ModuleList([InteractionNetwork(

hidden\_size, 3

) **for** \_ **in** range(n\_mp\_layers)])

self**.**reset\_parameters()

**def** reset\_parameters(self):

torch**.**nn**.**init**.**xavier\_uniform\_(self**.**embed\_type**.**weight)

**def** forward(self, data):

*# pre-processing*

*# node feature: combine categorial feature data.x and contiguous feature data.pos.*

node\_feature **=** torch**.**cat((self**.**embed\_type(data**.**x), data**.**pos), dim**=-**1)

node\_feature **=** self**.**node\_in(node\_feature)

edge\_feature **=** self**.**edge\_in(data**.**edge\_attr)

*# stack of GNN layers*

**for** i **in** range(self**.**n\_mp\_layers):

node\_feature, edge\_feature **=** self**.**layers[i](node\_feature, data**.**edge\_index, edge\_feature**=**edge\_feature)

*# post-processing*

out **=** self**.**node\_out(node\_feature)

**return** out

## Training

Before we start training model, let's configure hyperparameters! Since accessible computaion power is limited in Colab, we will only run 1 epoch of training, which takes about 1.5 hour. Consequently, we won't be able to produce as accurate results as shown in original paper in this Colab. Alternatively, we provide a checkpoint of training model on entire WaterDrop dataset for 5 epochs, which takes about 14 hours with a GeForce RTX 3080 Ti.

In [ ]:

data\_path **=** OUTPUT\_DIR

model\_path **=** os**.**path**.**join("temp", "models", DATASET\_NAME)

rollout\_path **=** os**.**path**.**join("temp", "rollouts", DATASET\_NAME)

**!**mkdir -p "$model\_path"

**!**mkdir -p "$rollout\_path"

params **=** {

"epoch": 1,

"batch\_size": 4,

"lr": 1e-4,

"noise": 3e-4,

"save\_interval": 1000,

"eval\_interval": 1000,

"rollout\_interval": 200000,

}

Below are some helper functions for evaluation.

In [ ]:

**def** rollout(model, data, metadata, noise\_std):

device **=** next(model**.**parameters())**.**device

model**.**eval()

window\_size **=** model**.**window\_size **+** 1

total\_time **=** data["position"]**.**size(0)

traj **=** data["position"][:window\_size]

traj **=** traj**.**permute(1, 0, 2)

particle\_type **=** data["particle\_type"]

**for** time **in** range(total\_time **-** window\_size):

**with** torch**.**no\_grad():

graph **=** preprocess(particle\_type, traj[:, **-**window\_size:], **None**, metadata, 0.0)

graph **=** graph**.**to(device)

acceleration **=** model(graph)**.**cpu()

acceleration **=** acceleration **\*** torch**.**sqrt(torch**.**tensor(metadata["acc\_std"]) **\*\*** 2 **+** noise\_std **\*\*** 2) **+** torch**.**tensor(metadata["acc\_mean"])

recent\_position **=** traj[:, **-**1]

recent\_velocity **=** recent\_position **-** traj[:, **-**2]

new\_velocity **=** recent\_velocity **+** acceleration

new\_position **=** recent\_position **+** new\_velocity

traj **=** torch**.**cat((traj, new\_position**.**unsqueeze(1)), dim**=**1)

**return** traj

**def** oneStepMSE(simulator, dataloader, metadata, noise):

"""Returns two values, loss and MSE"""

total\_loss **=** 0.0

total\_mse **=** 0.0

batch\_count **=** 0

simulator**.**eval()

**with** torch**.**no\_grad():

scale **=** torch**.**sqrt(torch**.**tensor(metadata["acc\_std"]) **\*\*** 2 **+** noise **\*\*** 2)**.**cuda()

**for** data **in** valid\_loader:

data **=** data**.**cuda()

pred **=** simulator(data)

mse **=** ((pred **-** data**.**y) **\*** scale) **\*\*** 2

mse **=** mse**.**sum(dim**=-**1)**.**mean()

loss **=** ((pred **-** data**.**y) **\*\*** 2)**.**mean()

total\_mse **+=** mse**.**item()

total\_loss **+=** loss**.**item()

batch\_count **+=** 1

**return** total\_loss **/** batch\_count, total\_mse **/** batch\_count

**def** rolloutMSE(simulator, dataset, noise):

total\_loss **=** 0.0

batch\_count **=** 0

simulator**.**eval()

**with** torch**.**no\_grad():

**for** rollout\_data **in** dataset:

rollout\_out **=** rollout(simulator, rollout\_data, dataset**.**metadata, noise)

rollout\_out **=** rollout\_out**.**permute(1, 0, 2)

loss **=** (rollout\_out **-** rollout\_data["position"]) **\*\*** 2

loss **=** loss**.**sum(dim**=-**1)**.**mean()

total\_loss **+=** loss**.**item()

batch\_count **+=** 1

**return** total\_loss **/** batch\_count

Here is main training loop!

In [ ]:

**from** tqdm **import** tqdm

**def** train(params, simulator, train\_loader, valid\_loader, valid\_rollout\_dataset):

loss\_fn **=** torch**.**nn**.**MSELoss()

optimizer **=** torch**.**optim**.**Adam(simulator**.**parameters(), lr**=**params["lr"])

scheduler **=** torch**.**optim**.**lr\_scheduler**.**ExponentialLR(optimizer, gamma**=**0.1 **\*\*** (1 **/** 5e6))

*# recording loss curve*

train\_loss\_list **=** []

eval\_loss\_list **=** []

onestep\_mse\_list **=** []

rollout\_mse\_list **=** []

total\_step **=** 0

**for** i **in** range(params["epoch"]):

simulator**.**train()

progress\_bar **=** tqdm(train\_loader, desc**=**f"Epoch {i}")

total\_loss **=** 0

batch\_count **=** 0

**for** data **in** progress\_bar:

optimizer**.**zero\_grad()

data **=** data**.**cuda()

pred **=** simulator(data)

loss **=** loss\_fn(pred, data**.**y)

loss**.**backward()

optimizer**.**step()

scheduler**.**step()

total\_loss **+=** loss**.**item()

batch\_count **+=** 1

progress\_bar**.**set\_postfix({"loss": loss**.**item(), "avg\_loss": total\_loss **/** batch\_count, "lr": optimizer**.**param\_groups[0]["lr"]})

total\_step **+=** 1

train\_loss\_list**.**append((total\_step, loss**.**item()))

*# evaluation*

**if** total\_step **%** params["eval\_interval"] **==** 0:

simulator**.**eval()

eval\_loss, onestep\_mse **=** oneStepMSE(simulator, valid\_loader, valid\_dataset**.**metadata, params["noise"])

eval\_loss\_list**.**append((total\_step, eval\_loss))

onestep\_mse\_list**.**append((total\_step, onestep\_mse))

tqdm**.**write(f"\nEval: Loss: {eval\_loss}, One Step MSE: {onestep\_mse}")

simulator**.**train()

*# do rollout on valid set*

**if** total\_step **%** params["rollout\_interval"] **==** 0:

simulator**.**eval()

rollout\_mse **=** rolloutMSE(simulator, valid\_rollout\_dataset, params["noise"])

rollout\_mse\_list**.**append((total\_step, rollout\_mse))

tqdm**.**write(f"\nEval: Rollout MSE: {rollout\_mse}")

simulator**.**train()

*# save model*

**if** total\_step **%** params["save\_interval"] **==** 0:

torch**.**save(

{

"model": simulator**.**state\_dict(),

"optimizer": optimizer**.**state\_dict(),

"scheduler": scheduler**.**state\_dict(),

},

os**.**path**.**join(model\_path, f"checkpoint\_{total\_step}.pt")

)

**return** train\_loss\_list, eval\_loss\_list, onestep\_mse\_list, rollout\_mse\_list

Finally, let's load dataset and train model! It takes roughly 1.5 hour to run this block on Colab with default parameters. **If you are impatient, we highly recommend you to skip next 2 blocks and load checkpoint we provided to save some time; otherwise, make a cup of tea/coffee and come back later to see results of training!**

In [ ]:

*# Training model is time-consuming. We highly recommend you to skip this block and load checkpoint in next block.*

*# load dataset*

train\_dataset **=** OneStepDataset(data\_path, "train", noise\_std**=**params["noise"])

valid\_dataset **=** OneStepDataset(data\_path, "valid", noise\_std**=**params["noise"])

train\_loader **=** pyg**.**loader**.**DataLoader(train\_dataset, batch\_size**=**params["batch\_size"], shuffle**=True**, pin\_memory**=True**, num\_workers**=**2)

valid\_loader **=** pyg**.**loader**.**DataLoader(valid\_dataset, batch\_size**=**params["batch\_size"], shuffle**=False**, pin\_memory**=True**, num\_workers**=**2)

valid\_rollout\_dataset **=** RolloutDataset(data\_path, "valid")

*# build model*

simulator **=** LearnedSimulator()

simulator **=** simulator**.**cuda()

*# train model*

train\_loss\_list, eval\_loss\_list, onestep\_mse\_list, rollout\_mse\_list **=** train(params, simulator, train\_loader, valid\_loader, valid\_rollout\_dataset)

Epoch 0: 4%|▍ | 1000/24875 [05:05<180:44:17, 27.25s/it, loss=1.18, avg\_loss=1.01, lr=0.0001]

Eval: Loss: 0.9509934263597347, One Step MSE: 1.8265435012861906e-07

Epoch 0: 8%|▊ | 2000/24875 [10:09<173:44:09, 27.34s/it, loss=0.935, avg\_loss=0.998, lr=9.99e-5]

Eval: Loss: 0.9384889955305976, One Step MSE: 1.8022839299283517e-07

Epoch 0: 12%|█▏ | 3000/24875 [15:14<165:58:02, 27.31s/it, loss=0.361, avg\_loss=0.932, lr=9.99e-5]

Eval: Loss: 0.3940979961749058, One Step MSE: 7.564177890267815e-08

Epoch 0: 16%|█▌ | 4000/24875 [20:16<158:19:53, 27.31s/it, loss=0.233, avg\_loss=0.786, lr=9.98e-5]

Eval: Loss: 0.2425671485986357, One Step MSE: 4.653181075323239e-08

Epoch 0: 20%|██ | 5000/24875 [25:20<149:13:58, 27.03s/it, loss=0.183, avg\_loss=0.681, lr=9.98e-5]

Eval: Loss: 0.19130304984604624, One Step MSE: 3.66730457891334e-08

Epoch 0: 24%|██▍ | 6000/24875 [30:22<141:45:03, 27.04s/it, loss=0.209, avg\_loss=0.612, lr=9.97e-5]

Eval: Loss: 0.16885312529238858, One Step MSE: 3.238040706305506e-08

Epoch 0: 28%|██▊ | 7000/24875 [35:23<134:12:25, 27.03s/it, loss=0.217, avg\_loss=0.555, lr=9.97e-5]

Eval: Loss: 0.15328593740173857, One Step MSE: 2.939762471182421e-08

Epoch 0: 32%|███▏ | 8000/24875 [40:25<126:35:01, 27.00s/it, loss=0.117, avg\_loss=0.511, lr=9.96e-5]

Eval: Loss: 0.14538169509680324, One Step MSE: 2.7891650137054557e-08

Epoch 0: 36%|███▌ | 9000/24875 [45:25<119:09:52, 27.02s/it, loss=0.216, avg\_loss=0.475, lr=9.96e-5]

Eval: Loss: 0.15156149027503762, One Step MSE: 2.9048361519680295e-08

Epoch 0: 40%|████ | 10000/24875 [50:28<111:40:34, 27.03s/it, loss=0.19, avg\_loss=0.445, lr=9.95e-5]

Eval: Loss: 0.1281800843008653, One Step MSE: 2.4587091128342988e-08

Epoch 0: 44%|████▍ | 11000/24875 [55:31<103:59:57, 26.98s/it, loss=0.122, avg\_loss=0.42, lr=9.95e-5]

Eval: Loss: 0.1261842829831112, One Step MSE: 2.419977349414354e-08

Epoch 0: 48%|████▊ | 12000/24875 [1:00:34<96:45:36, 27.06s/it, loss=0.155, avg\_loss=0.401, lr=9.94e-5]

Eval: Loss: 0.14466826888430157, One Step MSE: 2.7776733302164156e-08

Epoch 0: 52%|█████▏ | 13000/24875 [1:05:37<89:44:00, 27.20s/it, loss=0.137, avg\_loss=0.383, lr=9.94e-5]

Eval: Loss: 0.12485936950065699, One Step MSE: 2.3949403594268645e-08

Epoch 0: 56%|█████▋ | 14000/24875 [1:10:40<82:00:18, 27.15s/it, loss=0.167, avg\_loss=0.368, lr=9.94e-5]

Eval: Loss: 0.16495190527278128, One Step MSE: 3.160211703184723e-08

Epoch 0: 60%|██████ | 15000/24875 [1:15:44<74:31:35, 27.17s/it, loss=0.151, avg\_loss=0.354, lr=9.93e-5]

Eval: Loss: 0.1276598329389306, One Step MSE: 2.4446601871816098e-08

Epoch 0: 64%|██████▍ | 16000/24875 [1:20:50<67:03:04, 27.20s/it, loss=0.198, avg\_loss=0.342, lr=9.93e-5]

Eval: Loss: 0.12988546975795479, One Step MSE: 2.4881985310299766e-08

Epoch 0: 68%|██████▊ | 17000/24875 [1:25:56<59:30:17, 27.20s/it, loss=0.0908, avg\_loss=0.33, lr=9.92e-5]

Eval: Loss: 0.12723309360145085, One Step MSE: 2.4334121977178723e-08

Epoch 0: 72%|███████▏ | 18000/24875 [1:31:02<52:00:08, 27.23s/it, loss=0.171, avg\_loss=0.32, lr=9.92e-5]

Eval: Loss: 0.1620473044825616, One Step MSE: 3.119730222137828e-08

Epoch 0: 76%|███████▋ | 19000/24875 [1:36:10<44:26:09, 27.23s/it, loss=0.11, avg\_loss=0.31, lr=9.91e-5]

Eval: Loss: 0.11340397444233251, One Step MSE: 2.1779060378654464e-08

Epoch 0: 80%|████████ | 20000/24875 [1:41:15<36:51:11, 27.21s/it, loss=0.217, avg\_loss=0.302, lr=9.91e-5]

Eval: Loss: 0.1196164065025675, One Step MSE: 2.2998157158592724e-08

Epoch 0: 84%|████████▍ | 21000/24875 [1:46:25<29:19:51, 27.25s/it, loss=0.116, avg\_loss=0.294, lr=9.9e-5]

Eval: Loss: 0.108936554731328, One Step MSE: 2.0906139517657764e-08

Epoch 0: 88%|████████▊ | 22000/24875 [1:51:30<21:44:17, 27.22s/it, loss=0.299, avg\_loss=0.286, lr=9.9e-5]

Eval: Loss: 0.12898710272439615, One Step MSE: 2.4827022738471204e-08

Epoch 0: 92%|█████████▏| 23000/24875 [1:56:35<14:10:06, 27.20s/it, loss=0.145, avg\_loss=0.279, lr=9.89e-5]

Eval: Loss: 0.08806178554401424, One Step MSE: 1.688884021522341e-08

Epoch 0: 96%|█████████▋| 24000/24875 [2:01:39<6:36:34, 27.19s/it, loss=0.128, avg\_loss=0.273, lr=9.89e-5]

Eval: Loss: 0.09211417896990512, One Step MSE: 1.7663254729517782e-08

Epoch 0: 100%|██████████| 24875/24875 [2:04:48<00:00, 3.32it/s, loss=0.0777, avg\_loss=0.267, lr=9.89e-5]

In [ ]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

*# visualize loss curve*

plt**.**figure()

plt**.**plot(**\***zip(**\***train\_loss\_list), label**=**"train")

plt**.**plot(**\***zip(**\***eval\_loss\_list), label**=**"valid")

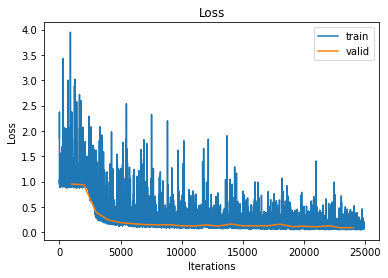
plt**.**xlabel('Iterations')

plt**.**ylabel('Loss')

plt**.**title('Loss')

plt**.**legend()

plt**.**show()



Load checkpoint trained by us. Do **not** run this block if you have trained your model in previous block.

In [ ]:

simulator **=** LearnedSimulator()

simulator **=** simulator**.**cuda()

**!**wget -O temp/models/WaterDrop\_checkpoint.pt https://storage.googleapis.com/cs224w\_course\_project\_dataset/Checkpoints/WaterDrop\_checkpoint.pt

checkpoint **=** torch**.**load("temp/models/WaterDrop\_checkpoint.pt")

simulator**.**load\_state\_dict(checkpoint["model"])

## Visualization

Since video is 1000 frames long, it might take a few minutes to rollout.

In [ ]:

rollout\_dataset **=** RolloutDataset(data\_path, "valid")

simulator**.**eval()

rollout\_data **=** rollout\_dataset[0]

rollout\_out **=** rollout(simulator, rollout\_data, rollout\_dataset**.**metadata, params["noise"])

rollout\_out **=** rollout\_out**.**permute(1, 0, 2)

In [ ]:

**%matplotlib** inline

**import** matplotlib.pyplot **as** plt

**from** matplotlib **import** animation

**from** IPython.display **import** HTML

TYPE\_TO\_COLOR **=** {

3: "black",

0: "green",

7: "magenta",

6: "gold",

5: "blue",

}

**def** visualize\_prepare(ax, particle\_type, position, metadata):

bounds **=** metadata["bounds"]

ax**.**set\_xlim(bounds[0][0], bounds[0][1])

ax**.**set\_ylim(bounds[1][0], bounds[1][1])

ax**.**set\_xticks([])

ax**.**set\_yticks([])

ax**.**set\_aspect(1.0)

points **=** {type\_: ax**.**plot([], [], "o", ms**=**2, color**=**color)[0] **for** type\_, color **in** TYPE\_TO\_COLOR**.**items()}

**return** ax, position, points

**def** visualize\_pair(particle\_type, position\_pred, position\_gt, metadata):

fig, axes **=** plt**.**subplots(1, 2, figsize**=**(10, 5))

plot\_info **=** [

visualize\_prepare(axes[0], particle\_type, position\_gt, metadata),

visualize\_prepare(axes[1], particle\_type, position\_pred, metadata),

]

axes[0]**.**set\_title("Ground truth")

axes[1]**.**set\_title("Prediction")

plt**.**close()

**def** update(step\_i):

outputs **=** []

**for** \_, position, points **in** plot\_info:

**for** type\_, line **in** points**.**items():

mask **=** particle\_type **==** type\_

line**.**set\_data(position[step\_i, mask, 0], position[step\_i, mask, 1])

outputs**.**append(line)

**return** outputs

**return** animation**.**FuncAnimation(fig, update, frames**=**np**.**arange(0, position\_gt**.**size(0)), interval**=**10, blit**=True**)

anim **=** visualize\_pair(rollout\_data["particle\_type"], rollout\_out, rollout\_data["position"], rollout\_dataset**.**metadata)

HTML(anim**.**to\_html5\_video())