Iterative Expansion of Recurrent Graph Neural Networks

Peter Racioppo
Department of Computer Science
University of California, Los Angeles
277 Engineering VI, Los Angeles, CA, United States
pcracioppo@ucla.edu

***Abstract*—Graph neural networks (GNNs) are a class of neural network architectures used for modeling data with a graph structure. They have been very useful in problems involving relational data, including characterization of molecules, protein folding, path planning, and physics simulation and control. For many of these problems, a static graph structure is sufficient, but when the graph structure changes over time, it becomes necessary to update nodes and edges dynamically. The extension of GNNs to this setting is referred to as Dynamic GNNs. In this paper, we review important models of dynamic GNNs and their application to a variety of important problems, including physics simulation, network modeling, and population interaction.**

***Index Terms*—Graph neural networks, interaction networks, graph networks, message passing, network modeling**

I. A Review of Dynamic Neural Networks for Modeling Physics

A. Introduction

The success of neural networks in modeling complex real-world data owes to three properties: (1) Neural networks are universal function approximators, meaning that a sufficiently large neural network can approximate any function (under mild smoothness conditions) to arbitrary accuracy; (2) They represent functions in a distributed, hierarchical fashion, and their expressiveness increases exponentially with the number of layers. This helps deal with the difficulty of modeling high-dimensional data, the so-called "curse of dimensionality"; (3) They are differentiable and can therefore be trained end-to-end with gradient descent.

In principle, these three properties are sufficient to learn extremely complex relationships, but in practice, the most successful use cases have been limited to sequential data (i.e. natural language) and data arranged in a lattice (i.e. natural images). For analyzing image data, convolutional neural networks (CNNs) have surpassed the performance of fully-connected networks by introducing several priors influenced by biological vision, which dramatically decrease the size of the parameter space over which the network must be optimized: CNNs assume local patterns of network connectivity, pooling over local features, and parameter sharing across the network. In natural language processing, the state of the art is achieved by Transformer networks, which incorporate an "attention" mechanism in which the network can learn which relations between data are likely to be most important.

Many classes of real-world problems involve data with more complex structure. Graphs are a powerful and general tool for modeling problems which involve complex sets of relations between discrete entities. It is therefore natural to ask whether neural networks can be generalized from the setting of sequence or lattice data to deal with graph data with less regular structure. Several difficulties present themselves: The most obvious approach to feeding graph data into a neural network is to flatten the node features into a one dimensional vector, but this removes the information about the connectivity and edge features of the graph. Moreover, graphs lack a well-defined ordering over nodes, which means that a neural network trained on vectorized data from a graph will lack generalization ability to a new graph (since the vectors of nodes features will have different orderings for the two graphs). Analogously to CNNs, one can define a graph convolution as an aggregation of node features over the neighborhood of a particular node. Figure 1 illustrates the GNN pipeline: graph data, including node and edge features, is passed through a series of graph convolutions, implemented using neural networks, eventually outputting predictions for node and edge features, new edges, or new graphs, subgraphs, or other global features.

Figure 2 illustrates a typical GNN architecture. Each node in the graph defines its own computation graph. Features from the node's neighbors are passed through a neural network, which transforms and aggregates them. This approach can be extended recursively: the features of the neighbors of each neighbor node are passed through their own neural networks, and so on. One can thus account for influences of nodes which are k-hops away in the graph. Thus, a GNN consists of not one neural network, but a collection of neural networks, one for each neighboring node, all of which are trained end-to-end. The depth of GNNs is limited by the fact that the number of neural networks increases quickly with the number of hops from the head node, and only a small number of hops are required to reach the large majority of nodes in most graphs. As in a CNN, the parameters of the neural networks are shared across the GNN, to improve generalization and the tractability of training.

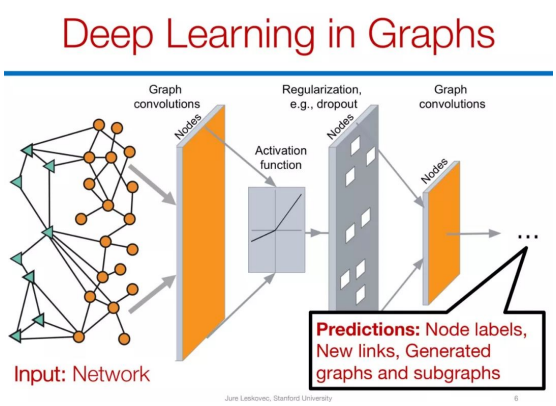

Figure 1: Forward pass of the graph network.

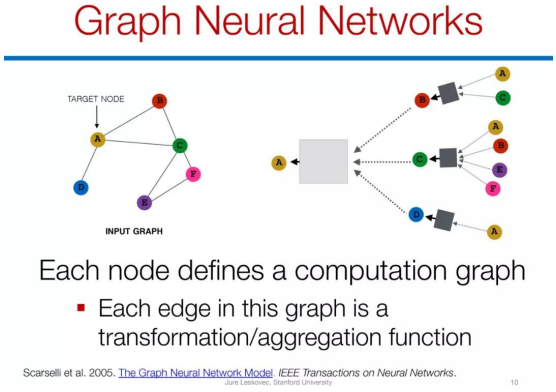

Figure 2: Forward pass of the graph network.

Note that the aggregation functions used in a GNN must be transitive, in order to be invariant to node ordering (needed for generalizability). A number of papers have experimented with aggregation functions such as summation, mean, and max. It has been shown that summation is the most expressive possible aggregation function.

Self-supervised learning is readily done in GNNs by removing edge or node features at random and training the network on whether it can predict them. GNN algorithms have been designed which can perform fast sub-graph matching and counting, and perform reasoning in knowledge graphs. GNNs have also been applied with great success to a number of important scientific problems involving relational data, including protein folding, physics simulation, drug discovery and design, logical inference, and epidemic modeling. We discuss one such application in greater detail in the following section.

B. Interaction Networks for Learning about Objects, Relations and Physics (2016)

The authors introduce the *interaction network* (IN), a neural network architecture they describe as the first "general-purpose, learnable physics engine." [1]

The interaction network draws inspiration from physics simulation: a system is represented as a series of objects, together with a set of relations between objects. Objects and relations are represented separately: an interaction effect at the $(t+1)$th time step between N_O object states can be written as $e_{t+1} = f_R(o_{1,t}, ..., o_{N_O,t}, r)$, a function of object states (i.e. position, velocity) $o_{i,t}$ at time t along with relation attributes r (i.e. a constant input such as a spring constant); the object state is updated as $o_{i,t+1} = f_O(o_{i,t}, e_{t+1})$.

The relations between object states can be represented as a graph $G = \langle O, R \rangle$, in which the nodes $O = \{o_j\}$, $j \in \{1, ..., N_O\}$ represent object states and the edges $R = \{\langle i, j, r_k \rangle_k\}$, $k \in \{1, ..., N_R\}$ represent the relations between them. Here, i and j index the nodes between which the kth edge exists, and r_k is an edge attribute. Representing relations and objects separately allows the network to generalize to new settings with different graph structures. (The authors, in fact, choose to represent the system as a multi-graph in order to separately account for different types of interactions, e.g. electromagnetic forces vs. rigid-body forces).

The interaction network takes as input $G = \langle O, R \rangle$, as well as $X = \{x_j\}$, , $j \in \{1, ..., N_O\}$, which represent forces (or control inputs) external to the system modeled by G. The interaction network is defined as follows:

$$IN(G) = \phi_O[a(G, X, \phi_r[m(G)])] \quad (1)$$

Here, m is a function that aggregates the input objects and relations into a standard form for interactions (in this case, through a matrix multipication), ϕ_r is a function that predicts the effect of each interaction, a is a function that aggregates the effects on each node, and ϕ_O is a function that updates the state of each node. The authors also explore passing the outputs of ϕ_O through another aggregation function and another nonlinear function, which returns a single output for the graph, in order to indicate whether the system as a whole meets certain criteria.

In order to be invariant to graph structure, the function a must be commutative and associative, i.e. a function such as summation, mean, or maximum (the authors of the present paper use a summation). The implementation of the interaction network described by the authors consists of aggregating local graph information using matrix multiplications and training ϕ_R and ϕ_O as multi-layer perceptrons (MLPs). The same f_R and f_O are used for each edge and node, which improves generalization. This parameter sharing is analogous to CNNs. The appearance of o_j in both ϕ_R and ϕ_O is analogous to skip connections in residual networks. The implementation described by the authors appears to be equivalent, or at least analogous, to a single-layer graph neural network, since it aggregates information from neighboring nodes (a single hop

in the graph).

The authors tested the interaction networks predictions of trajectories and potential energy for three types of physical system: n-body gravitational systems; balls bouncing in a box; and a string (a serial chain of rigid bodies joined by springs) colliding with a rigid body. They compared the results to a model which outputs the input velocities, an MLP which takes a flattened vector of all input data, and a variant of the IN with the ϕ_R (that is, the relational content) removed. The IN achieved test error rates (on both trajectory and potential energy predictions) which were far lower than these alternatives, and produced visually convincing simulated trajectories. It was able to simulate thousands of timesteps, despite being trained on single time-step predictions. The IN was able to generalize to systems with fewer or greater numbers of interacting bodies than the system on which it was trained.

The authors mention as a possible direction for future research the possibility of using the IN to perform model predictive control (MPC), since the system model is differentiable. They also discuss the need for a perceptual front-end that can interpret sensory information as graph data that can be input to the IN. Finally, they discuss the possibility of combining the IN with a recurrent neural network (RNN) or using the IN as a probabilistic generative model.

C. Graph networks as learnable physics engines for inference and control (2018)

In this work [2], the authors introduce *graph networks* (GNs), a generalization of the interaction networks introduced in (Battaglia et al, 2016). The authors begin the paper by observing that in order to model complex systems whose states or trajectories scale combinatorially, a powerful method is to represent a system as composed of discrete "objects" and "relations." The same computations are applied object-wise to the objects and relation-wise to the relations. The assumption here is that interactions between objects are determined by similar rules. One can therefore learn object and relation update rules and then generalize these rules to new configurations of objects.

The authors represent a physical system as a graph: objects are represented as nodes, and the relations between them as edges. (In a sense, the existence of an edge between two nodes is a weaker assumption than its non-existence, since the weights of an existing edge can be learned to be approximately zero, whereas the non-existence of an edge implies strong confidence about the lack of a relation.) The graph network architecture generalizes interaction networks in the following ways: (1) They include global representations of the state of the graph, in addition to node and edge-level representations. (2) They introduce a mechanism for mapping an input graph to an output graph with new node and edge features. This is used as a means of representing the evolution of a physical system from one time step to another. A GN forward pass consists of an edge-wise function f_e, a node-wise function f_n, and a global function f_g (given by multi-layer perceptrons (MLPs)), which map vectors of old node and edge features (n, e, g) and global graph features to new vectors **n,e** and **g**. The three functions are arranged hierarchically, as shown in the below figure: the edge-wise function f_e takes the edge, node, and global features **n,e** and **g** as inputs and outputs a new **e**; the node-wise function f_n takes as inputs f_e and **n** and **g** and outputs and new **n**; finally, the global function f_g takes as inputs f_e, f_n and **g** and outputs a new **g**. This process can be thought of as equivalent to message passing in the original graph.

The authors experiment with a number of architectures incorporating the GN forward pass as a basic unit. For prediction, the authors use an architecture with two GNs in sequence, with a skip connection across the first GN (the two graph outputs are then concatenated). This increases the model's expressiveness by allowing the node and edge features in the first layer to directly interact with the output of the effect of the first GN. The authors also introduce a model which passes edge, node, and global features through a recurrent neural network (RNN), in this case three GRUs, before passing the output through a GN. All architectures were trained to learn *changes* in states, and the new states are obtained by adding these to the previous states. The authors also create an inference model for system identification, which learns a GNN model for a physical system from a series of graphs representing that system's trajectory and relations at each time step. The authors take advantage of the fact that the GN model is differentiable to compute model predictive controllers (MPCs). The GN model is used to compute the trajectories that result from proposed action sequences (control inputs) and backpropagate gradients of a state-dependent reward signal with respect to the action sequence to the GN parameters. The authors also performed experiments to test the use of a GN in a model-based reinforcement learning algorithm.

The authors tested their models using seven different Mujoco simulation environments. Training data was generated by applying random controls to the systems and observing the trajectories. In experiments testing system identification, the parameters of the models were varied procedurally. The MPC model was compared to a differential programming algorithm which had access to ground truth system models. The authors found that the trained inference models for system identification were able to make accurate trajectory predictions and supported generalization to different initial states. The model predictive control models were able to successfully learn control inputs which produced produced sophisticated patterns of movement and achieved results comparable to a sophisticated planning algorithm which had

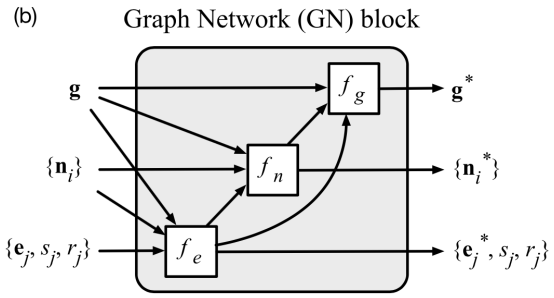

Figure 3: Forward pass of the graph network.

access to the ground truth system model, and a deep dynamic programming model. These results generalized to multiple systems, supported zero-shot generalization, and produced good results under partial observations. The authors jointly trained a GN and a reinforcement-learning policy-functinon. This model-based RL agent was able, in some cases, to achieve a higher-level of performance after fewer episodes than the baseline model-free RL agent. Future research directions include scaling up the system to real-world control problems, extending the model to stochastic systems, and performing system identification over the structure of the system (i.e. the graph structure), as opposed to only its parameters.

D. Learning to Simulate Complex Physics with Graph Networks (2020)

This paper [3] expands on the authors' previous papers on interaction networks and graph networks with a GNN-based "Graph network simulator" (GNS) for physics simulation. The method is based on the observation that physical interactions between particles can be viewed as message passing on a graph, with nodes representing particles and edge relations representing their interactions, e.g. exchanges of momentum and energy. The GNS model is composed of three parts: an Encoder, a Processor, and a Decoder (see figure.) The Encoder constructs a graph from a set of particles by assigning a node to each particle and creating edges between particles which are within a certain "connectivity radius," which is kept constant across all simulations of the same resolution. The Encoder assigns node and edge features as functions of particle states with the outputs of a multi-layer perceptron (MLP). The Processor simulates interactions between particles by computing M steps of message passing through the graph, using the Graph Network architecture discussed in the previous section, generating a sequence of latent graphs. The Decoder transforms the node features of the final latent graph output by the processor into dynamics information, again using an MLP; this information is then passed through an Euler integrator in order to update particle states.

The authors tested their model on three regimes of particle

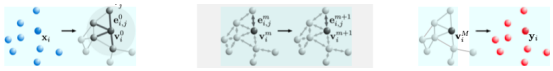

Figure 4: Dynamic Graph Generation

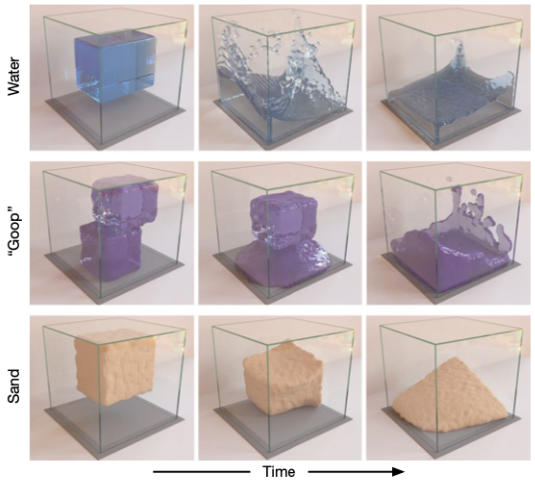

Figure 5: Dynamic Graph Generation

interaction: water (which exhibits little damping); "goop," a viscous, non-Newtonian material; and sand (which exhibits complex frictional interactions) (simulation experiments for these three materials are shown in the figure). The prediction targets in all cases are the per-particle average acceleration per time step. The authors found that adding random walk noise to the training data significantly improved robustness. The authors atested the model's generalization ability by evaluating the model on initial conditions not seen during training, different numbers of particles, different configurations of obstacles, etc. The much different behaviors exhibited by these materials normally requires separate physics simulations, but the GNS model was able to learn all three regimes to high accuracy and resolution. The model was also able to generalize to much different settings than those encountered in training and to simulate interactions between different kinds of materials, with up to 19 thousand particles. Although the GNS models were trained to make single time step predictions, they were able to roll out plausible trajectories over thousands of time steps. The authors found that a greater number of message passing steps and higher connectivity radius significantly improved performance. Increasing these parameters allows for longer-range communication between nodes, but comes at great computational cost.

E. Hamiltonian Graph Networks with ODE Integrators (2019)

In this paper [4], the authors explore including inductive physical biases in dynamic GNNs in order to promote better predictive accuracy and to ensure that trained models obey physical laws such as energy conservation. The authors begin the paper by noting that GNN architecture can be thought

of as an inductive bias about the structure of the physical system being modeled, such as modeling particles as nodes and relations between them as edges (which biases the system toward learning more local effects).

The authors discuss three models: The Delta graph network (DeltaGN) replicates the approach in previous papers (such as in the last section). A graph network (GN) is used to predict changes in the modeled system's states. This is analogous to learning an integrator. The authors next introduce the ODE graph network (OGN), which instead learns the time derivatives of each particle. These are then passed through a Runge-Kutta integrator in order to update the states. This amounts to an inductive bias that the system dynamics can be modeled as first-order ODEs. Finally, the authors introduce the Hamiltonian ODE graph network (HOGN), which is trained to learn the Hamiltonian of the system (a scalar, usually corresponding to the energy of the system), which is then differentiated according to Hamilton's equations in order to obtain the time derivatives of generalized position $\boldsymbol{q}$ and momentum $\boldsymbol{p}$:

$$\begin{aligned} \frac{d\boldsymbol{q}}{dt} &= \frac{d\mathcal{H}}{dp} \\ \frac{d\boldsymbol{p}}{dt} &= -\frac{d\mathcal{H}}{dq} \end{aligned} \tag{2}$$

This is a stronger prior which imposes a global requirement in the computation of a single scalar property of the whole system.

The authors trained their models on a system of between 4 and 9 particles exerting spring forces on each other according to Hooke's Law. All models were trained to make next-step predictions of position and momentum. Using a fourth-order Runge-Kutta integrator, the authors found that the OGN and HOGN models had lower rollout error than the DeltaGN model. The average energy of system also stayed closer to the true values under these models than DeltaGN. However, for lower-order integrators, the OGN and HOGN models actually displayed higher trajectory and energy errors than did the DeltaGN mode. The authors speculate that this is because, since the non-Hamiltonian models are not constrained to impose Hamiltonian dynamics, they can learn more accurate approximations of time derivatives in order to offset the errors due to the lower-order integrators.

Using a third-order integrator, the average energy predicted by the HOGN model was closer to the ground truth than for the OGN model. Both the OGN and HOGN models displayed better zero-shot generalization ability than DeltaGN. It was also found that the HOGN model generalized much better to different integrators than did the OGN model.

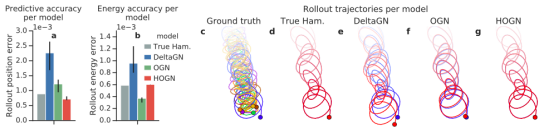

Figure 6: Energy and trajectory results produced by the DeltaGN, OGN, and HOGN models.

II. Design of a Recurrent Graph Convolutional Network

This section details the design and implementation of a recurrent graph convolutional network. A recurrent (or dynamic) graph neural network (RGNN) learns the state of node x_i at time t as a first-order ordinary differential equation (ODE) $\dot{x}_i(t) = f_i(x_1, ..., x_n)$, where each f_i is a nonlinear equation of the other states. In general, our dynamics may be of higher order. In this case, the RGNN learns a first-order approximation for the higher-order dynamics. It would be preferable for the network to learn the correct higher-order representation of the system, in which each state that appears in the equations of motion is represented by its own state variable, and hence node. However, in general, we do not know the structure of the set of coupled differential equations governing the system, hence the need for a neural network.

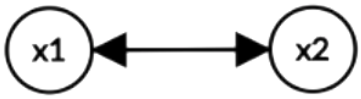

Figure 7: Architecture of a fully connected graph

Let us instead suppose that the system in question has second-order dynamics (as is the case for most physics). The equations of motion are then of the form $\ddot{x}_i(t) = f_i(x_1, ..., x_n, \dot{x}_1, ..., \dot{x}_n)$. In order to get this second-order system into the first-order form $\dot{x}_i(t) = f_i(x_1, ..., x_n)$, we can introduce a set of n new variables $\{v_1, ..., v_n\}$ that represent the derivatives of $\{x_1, ..., x_n\}$. Thus, a system of n 2nd order ODEs can be rewritten as a set of $2n$ 1st order ODEs.

$$\begin{aligned} f_1(x_1, ..., x_n, v_1, ..., v_n) &= \dot{v}_1 \\ v_1 &= \dot{x}_1 \\ &\vdots \\ f_n(x_1, ..., x_n, v_1, ..., v_n) &= \dot{v}_n \\ v_n &= \dot{x}_n \end{aligned} \tag{3}$$

Or in matrix, form: $f(X) = \dot{X}$. The state variables obey the Markov relationships: $(x_{i,t+1}|v_{i,t}, x_{i,t}) \perp\!\!\!\perp (x_{j,t}, v_{j,t}), \forall i \neq j$, which specify that the x_i nodes affect each other only through their respective v_i nodes.

The above procedure is summarized in the following

algorithm:

Algorithm 1: Graph Expansion

Inputs: Nodes and edges of the original graph
Outputs: Nodes and edges of the expanded graph
$N =$ total number of nodes
Add N new nodes to graph
Add edges to graph, following Markov assumptions:
$(x_{i,t+1}|v_{i,t}, x_{i,t}) \perp\!\!\!\perp (x_{j,t}, v_{j,t}), \forall i \neq j$

We will call the GNN representing this new system the *expanded graph*. An example is shown in Figure 8 for an original graph with two nodes x_1 and x_2. The expanded graph has two new nodes v_1 and v_2, and edges obeying the Markov assumptions. Self-loops are not pictured.

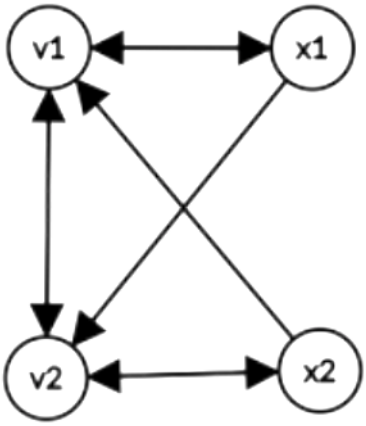

Figure 8: Architecture of an expanded graph

We can continue expanding the graph to represent higher-order dynamics. To represent an ODE of order k, we need kn nodes and $(1+\sum_{i=1}^{k}(i-1))n^2+(k-1)n = (k+\sum_{i=2}^{k}(i-2))n^2+(k-1)n = (\frac{1}{2}k(k-1)+1)n^2+(k-1)n = O((kn)^2)$ edges.

More generally, if the first-order graph has n nodes and e edges, with $n \leq e \leq n^2$ edges, then to represent an ODE of order k (with the same e for each derivative order), we need kn nodes and $(1 + \sum_{i=1}^{k}(i-1))e + (k-1)n = (\frac{1}{2}k(k-1)+1)e + (k-1)n = O(k^2 e)$ edges. Thus, the number of edges is quadratic in the order of the ODE, but linear in the number of edges in the first-order graph, and the Markov assumptions reduce the number of edges in the graph by roughly half.

If we make the additional assumption that $(x_{i,t}^{(k)}|x_{i,t}^{(k-1)}) \perp\!\!\!\perp x_{i,t}^{(k-j)}$ for all $1 < j \leq k$, that is, that a kth derivative is independent of all higher-order derivatives given the $(k+1)$th derivative, then we only need $O(ke)$ edges. Since, for higher-order representations, the number of edges in the expanded graph is dominated by the edges between the new nodes, we can greatly reduce the number of edges in the expanded graph if we assume that the new nodes have the same connectivity pattern with each other as the original nodes had with each other. If the original graph was sparse, this radically reduces the number of edges in the expanded graph.

Note also that the edges from the v_i to x_i don't need to be learned, since we can simply use an integrator on these edges.

We can instead expand one-new node at a time. After training for a certain period, we evaluate the network's performance on a testing set, and select the node with the highest average error for expansion. Pseudocode for this algorithm is shown below:

Algorithm 2: Adaptive Graph Expansion

Inputs: Nodes and edges of the original graph, number of nodes to expand N_e
Outputs: Nodes and edges of the expanded graph
Train original GNN
Test GNN and find node with highest average loss
for *i in range(N_e)* **do**
    Expand node with highest average testing loss, according to the Markov assumptions:
    $(x_{i,t+1}|v_{i,t}, x_{i,t}) \perp\!\!\!\perp (x_{j,t}, v_{j,t}), \forall i \neq j$
    Train the expanded GNN
    Test the expanded GNN and find node with the highest testing loss, excluding the nodes that have already been expanded

There are several potential advantages to the methods outlined above: The graph structure imposes a stricter prior on the dynamics by forcing the network to learn the state variables and their derivatives separately. If, for instance, the dynamics are truly first-order, than the expanded GNN need only learn the true first-order transformations at each edge rather than first-order approximations of higher-order dynamics. In this case, we can use our trained GNN to produce more accurate closed-form representations of the true dynamics, which may be useful for analysis, interpretability, or control.

Algorithms 1 and 2 were implemented using the Pytorch Geometric Temporal library, a temporal extension of the Pytroch Geometric library for GNNs. [5]. The network design is a single layer of the Diffusion Convolutional Gated Recurrent Unit introduced in [6], followed by a ReLU activation and a linear layer. The loss is a Mean-Square Error loss averaged over nodes and time-steps. The data was trained on the built in Chickenpox dataset, which according to the

Pytorch Geometric Temporal documentation is "a dataset of county-level chicken pox cases in Hungary between 2004 and 2014."

Results of training with Algorithm 1 (graph expansion) are shown in Figure 9.

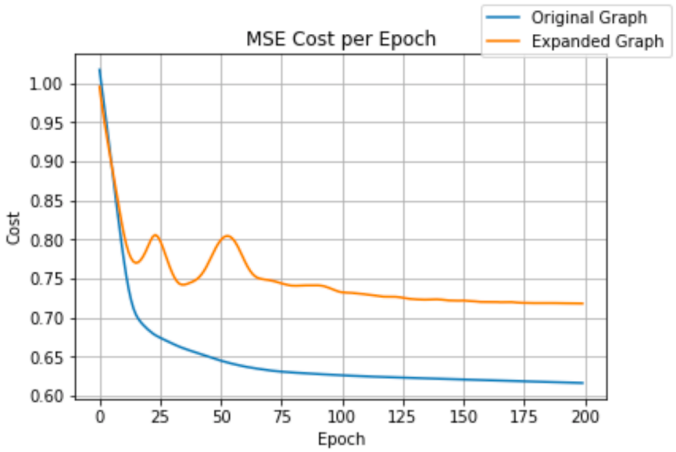

Figure 9: Results of the graph expansion algorithm.

Notice that the cost per epoch curve of the expanded graph is not monotonically decreasing. This is because were are training on all of the nodes in the expanded graph, but only evaluating the performance of the original nodes. Expanding the graph increases the training loss. This is to be expected since the expansion process adds new edges to the graph and hence increases the length of the training process. Future work should refine the algorithm and test whether training and testing error are eventually better than in the unexpanded graph.

Results for the Adaptive GNN algorithm are shown in Fig. 10.

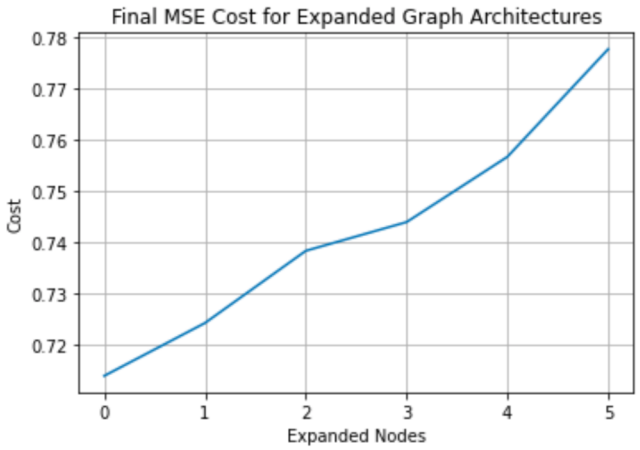

Figure 10: Results of the adaptive graph expansion algorithm.

Unfortunately, testing MSE error is a monotonic increasing function of the size of the graph, so the expansion procedure actually worsens performance. However, this is again to be expected, since each expansion increases the number of parameters that need to be trained. Note that the number of training epochs is held constant for each model, but the larger models should need longer to train. Simply training each model more might give the desired results, but this will have to wait for future work.

Two sampling methods were also implemented to sample a subgraph at each training step: (1) Simply dropping k random edges from the graph and (2) Using a random walk of length num_hops, starting from a random node and moving between nodes with probability proportional to the weights of the adjacent edges. Results are shown below.

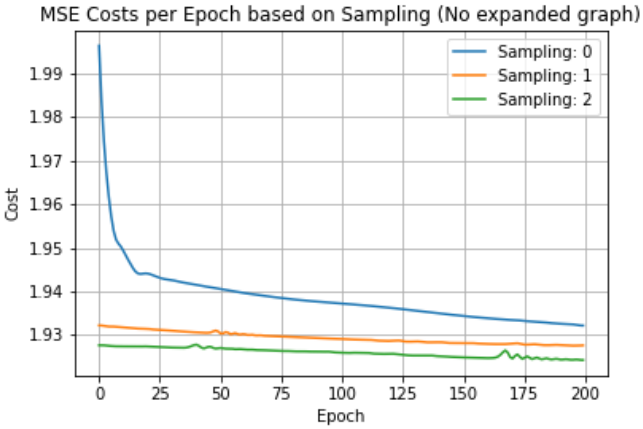

Figure 11: Results of RGNN with various sampling methods.

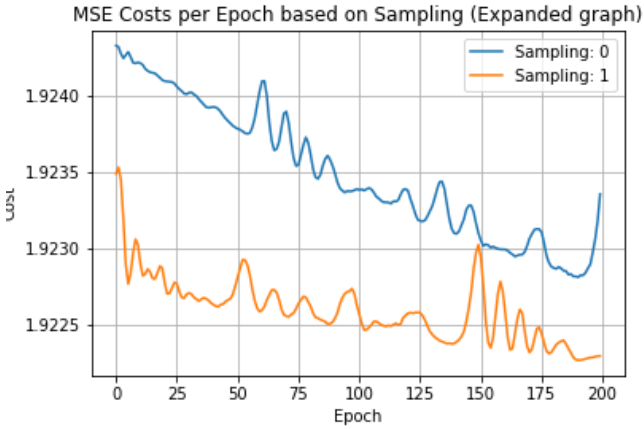

Figure 12: Results of RGNN using an expanded graph with various sampling methods.

Although the RGNN and Adaptive Graph Expansion algorithm were both not implemented with our COVID dataset, the exploration provides insight on possible future directions of

the research and with more time, we would explore such directions as they show to have promising results in terms of temporal data.

III. GNNs for Two-Point Boundary Value Control

As we have seen, Recurrent GNNs are powerful, learnable models for complex interactions. A natural next step is to apply this powerful modeling tool to control. Since these models are differentiable, we can use these models to compute optimal controllers via gradient descent or to do model-based reinforcement learning, both of which were implemented in [2]. Recurrent GNNs may potentially become a general-purpose tool for system identification control of difficult problems at the frontiers of control theory. For instance, they may prove useful to the difficult class of problems dealing with boundary control of partial differential equations (PDEs), which cover a broad class of control problems for fluid dynamics, thermodynamics, electrodynamics, etc. (For example, suppose we want to control a fluid by shaking its container, or we want to induce certain vibrations on a membrane by oscillating the ring on which it's suspended, or we want to control the diffusion of heat through a solid medium by only applying heat to its boundary, or we want to stabilize plasma in a fusion reactor using magnetic fields, etc.) Such methods might also be used to learn a model on the effect of pandemic interventions, and then compute optimal interventions, e.g. optimal quarantine scheduling in order to minimize a function of deaths and other social costs. However, such methods would lack any stability guarantees, due to the lack of closed-form representations for system dynamics from our learned model (note that the graph expansion method outlined in a previous section may be helpful in this regard).

We now outline a method for the use of GNNs for two-point boundary value control problems, those problems in which both the initial and final conditions of the system must meet constraints. Much of control theory deals with linear, first-order ODEs, which can be written in the form $\dot{X} = AX$.

Suppose we have a dynamical system of the form:
$\dot{x}(t) = f(x(t), u(t), t)$,
$x(t_0) = x_0$

and we wish to minimize a performance criterion of the form:
$J(u(\cdot); x_0) = \phi(x(t_f)) + \int_{t_0}^{t_f} \mathcal{L}(x(t), u(t), t)dt$, where $\mathcal{L}$ is the Lagrangian.

The optimal control $u^o(t)$ is given by Pontryagin's Weak Necessary Condition:

$$\frac{\partial}{\partial u} H(x^o(t), u^o(t), \lambda(t), t) = 0, \tag{4}$$

$$-\dot{\lambda}^T(t) = \frac{\partial}{\partial x} H(x^o(t), u^o(t), \lambda(t), t), \tag{5}$$

$$\lambda^T(t_f) = \phi_x(x(t_f)) \tag{6}$$

An algorithm which exploits Pontryagin's Weak Necessary Condition to find an optimal control $u^o(t)$ is outlined below:

Algorithm 3: Steepest Descent
(Modified from Speyer & Jacobson, 2010)

Inputs: Constants δ and ϵ, array of initial control u_1 (for each time step)
Outputs: Array of optimal control u^o for each time step.
while $||\frac{\partial}{\partial u} H(x^o(t), u^o(t), \lambda(t), t)|| > \delta$ **do**
  Integrate the system dynamics from t_0 to t_f to obtain the state path x_i.
  Integrate the adjoint λ_i equation backward along the nominal path.
  Update the nominal control:
  $u_{i+1}(t) \leftarrow u_i(t) - \epsilon \frac{\partial}{\partial u} H^T(x_i(t), u_i(t), \lambda(t), t)$.

To implement this algorithm using a GNN to model our system dynamics, we need to be able to compute derivatives with respect to control inputs, integrate system dynamics and Lagrange multiplier dynamics, and compute Hamiltonians and Lagrangians. Clearly, we can take derivatives, since the whole model is differentiable. We can integrate forward and backward in time using a GN-based forward model, as in [2]. Furthermore, it is possible to build a GNN which cann compute the Lagrangian Hamiltonian, as shown in [4]. Thus, it should be possible to solve two-point boundary value problems using GNNs. Implementing this method will be a subject of future work.

References

[1] P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu, "Interaction networks for learning about objects, relations and physics," 2016.

[2] A. Sanchez-Gonzalez, N. Heess, J. T. Springenberg, J. Merel, M. Riedmiller, R. Hadsell, and P. Battaglia, "Graph networks as learnable physics engines for inference and control," 2018.

[3] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. W. Battaglia, "Learning to simulate complex physics with graph networks," 2020.

[4] A. Sanchez-Gonzalez, V. Bapst, K. Cranmer, and P. Battaglia, "Hamiltonian graph networks with ode integrators," 2019.

[5] B. Rozemberczki, P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, , G. Lopez, N. Collignon, and R. Sarkar, "PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models," in *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, 2021, p. 4564–4573.

[6] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," 2018.