Stake-Free Evaluation of Graph Networks for Spatio-Temporal Processes

Erich Merrill *merriler@oregonstate.edu*
Oregon State University

Alan Fern *afern@oregonstate.edu*
Oregon State University

1 Abstract

Spatio-temporal processes are a class of prediction problem that are poorly served by traditional deep learning architectures owing to the problems' frequently irregular structures. We propose a method for encoding such prediction problems as a graph which describes both the relationship between input samples from the process and the desired prediction targets requested of the model. However, it is difficult to determine which graph neural network (GNN) architecture is appropriate to apply to these problem domains, as works proposing novel GNNs generally only evaluate them in settings where they are expected to perform optimally rather than investigating their behavior and properties in general. We aim to address this gap in knowledge by performing a fair, stake-free evaluation of three different GNN architectures on three distinct spatio-temporal problems. The goal of this stake-free evaluation is to examine the behavior of each model on each problem type rather than demonstrate that one dominates the other. We find that GNNs which cannot exploit edge features perform poorly in this setting, and that GNNs which learn explicit interpretable weight functions are slightly outperformed by their counterparts that employ black-box function for the same purpose. Finally, we release the software platform used to perform this evaluation, which is designed to enable practitioners to easily reproduce or extend our experiments.

2 Introduction

Many interesting real-world problems can be described as spatio-temporal processes. A spatio-temporal process problem consists of a set of data samples for which each sample has an associated timestamp and associated position in some shared space. Frequently, these samples represent discrete observations of some underlying continuous process of interest. Examples include a citizen science setting, in which a collection of data is gathered from a set of sensors distributed throughout some area that record environmental conditions over time; and a military engagement, in which a changing set of allied units periodically report their status and location, as well as any information available about the enemy's status and location. In both settings, the information of interest changes continuously over time, but generally its associated data would only be recorded at a set of instantaneous time points. In the environmental citizen science setting, the environmental conditions of interest likely change continuously throughout space as well, but we are only able to observe its state at the provided sensors' locations which we may not be able to control. Making useful inferences about the state of such processes requires jointly reasoning about both the spatial and temporal relationships between its samples.

There are well-known deep learning network architectures designed to process spatial and time series data: convolutional neural networks (CNNs) and recurrent neural networks (RNNs) respectively. These building blocks have previously been combined in various ways to construct models which are capable of reasoning about spatio-temporal data, such as by Shi et al. (2015) or Li et al. (2017) However, these architectures are challenging to effectively apply to real-world spatio-temporal processes in general. These architectures require that their input consists of a regular, discrete structure such as a grid of pixels for CNNs, and a sequence of predictably-spaced observations for RNNs. However, real-world spatio-temporal process data

are unlikely to conform to this 'grid world' paradigm. For example, in the citizen science setting the sensors' positions cannot be controlled and generally cannot be approximated with a regular grid structure, making it difficult to effectively apply CNNs to reason about the data's spatial relationships. Additionally these architectures require their input structure to be densely populated with data, meaning they are unable to natively handle sparse structures with 'missing' data entries. For example, in the military setting information about enemy activity may be extremely sparsely and irregularly distributed throughout the duration of the encounter, making it difficult to apply RNNs to infer the enemy units' complete state at any given point in time.

One solution to these issues is to re-sample the data such that it is forced to conform to the required grid structure. For example, one could re-sample a set of points in 2D space by imposing a grid structure over the same space, and assigning each cell in the grid a value based on the samples contained within its bounds. However, this approach has significant drawbacks. Performing this resampling by discretizing continuously-valued locations as described requires the grid resolution to be specifically tuned for each problem. If the grid resolution is too small data samples with close neighbors may be 'lost' since each grid cell only represents one sample, likely making it impossible for the model to have an accurate understanding of the state of the process. If the grid resolution is too large the majority of the cells will not represent any data sample in the problem, making it extremely difficult for the network to effectively reason about the relationships between the relevant data samples.

Another possible solution for some domains is to use an imputation model to fill in any data that is 'missing' from the required regular structure. However, in complex processes where the data samples are very sparsely distributed, the imputed samples far from any actual observations are unlikely to be informative. As before, the model will struggle to make useful inferences about the process if the vast majority of the samples in the sequence are 'fake' imputed samples. Additionally, generating and learning over these imputed samples may itself be significantly expensive. In these complex processes, computing a large number of imputed samples may impose an unacceptable performance overhead just to determine their values. On top of that, the model itself may require significantly more memory or other resources to process such large input data. Since the only purpose of the imputation is to allow the data to conform to the model's prescribed regular structure, a model that does not require such a regular structure would be more efficient since it is not required to generate this extra data, and presumably more performant since it can exclusively reason about the most informative samples rather than mostly working with the output of the imputation model.

These potential issues suggest that it would be preferable to employ a deep learning architecture that can directly consume irregularly-distributed, continuously-valued data such as that found in spatio-temporal processes. Graph neural networks (GNNs) are one such suitable architecture. Rather than operating on input of a fixed, static structure, graph networks directly exploit the relationships explicitly described by an arbitrary graph provided as input. Their use of parameter-sharing schemes and set functions allows them to consume graphs with arbitrary numbers of nodes, edges, and those with nodes of differing numbers of attached edges. This is much more flexible than the traditional CNN and RNN architectures, which are beholden to their discrete, fixed underlying structure. As a result, representing spatio-temporal problem instances with a graph structure and then processing them with a GNN model allows us to avoid the pitfalls describe above associated with applying CNN- and RNN-style models to real-world spatio-temporal processes.

Given the potential for graphs to represent sparse spatio-temporal data, there is a question of how effectively different GNN architectures will perform on graph representations of such problems. As graphs can represent extremely diverse types of data, different GNN architectures employ drastically different approaches to processing their input, with drastically different corresponding inductive biases meant to exploit different features of the provided graph structure. Each type of graph network is generally inspired by certain types of problems and often evaluations are limited to just those problem types. While such evaluations highlight strengths each model, they fail to provide insight into performance on other problem types. This makes it difficult for practitioners to infer which GNN architecture may be most appropriate for a problem at hand. This, for example, is the case for sparsely sampled spatio-temporal process. This lack of a useful, neutral examination of GNN models' capabilities suggests the need for a 'stake-free' evaluation of these models, in

Expression	Meaning
X	Set of all input samples that make up a problem instance
x	An individual sample within the process
Q	Set of all query targets that describe the problem instance's requested predictions
Q^*	Ground truth for queries Q
$Pos(x)$	The spatial location of the sample
$Time(x)$	The temporal location of the sample
$Feat(x)$	The sample's feature values, excluding positional information
$Ent(x)$	The sample's associated entity ID
$Dist_P(x, x')$	The difference vector between two samples' spatial locations
$Dist_T(x, x')$	The difference between samples' locations in time
$GNN(G, X)$	Calling a GNN to calculate latent encodings for the set of nodes X in graph G

Table 1: Spatio-Temporal Process Notation

which the goal is not to prove dominance of one model over the others. Rather, a 'stake-free' evaluation aims to examine the differing behavior and capabilities of each type of model on different problems.

In this work, we aim to address this gap in knowledge by fairly evaluating three meaningfully distinct GNN architectures on three different types of spatio-temporal problems. Each architecture is evaluated by defining three instantiations of each network type of varying sizes (i.e. Small, Medium, Large) such that every model of a given size has similar parameter count. To ensure that all models are trained and evaluated fairly we also propose a procedure for determining appropriate training hyperparameters for any given model instantiation, modified from the learning rate test proposed by Smith (2017).

We find that the edge-feature-aware graph networks significantly outperform the graph network architecture which exclusively relies on graph adjacency information to reason about the relationship between nodes. Of the edge-aware networks, some architecture learn explicit weight functions to filter the values of neighboring nodes. These weight functions can be visualized to demonstrate the model's understanding of the problem domain. However, the architectures that instead reason about each node's neighbors using a black-box function seem to generally perform better and exhibit better generalization performance despite their lack of explicit interpretability. Finally, we publicly release the software platform developed to perform all the experiments in this evaluation to allow others to reproduce or extend our work.

3 Spatio-Temporal Processes

Each instance of a spatio-temporal problem consists of a set of observations (or samples) from distinct entities that change over time while occupying some shared space, and a set of domain-specific queries which describe the desired predictions within that same space. It is important to note that there are no constraints on how an observation's associated position and time are described. Whereas most deep learning architectures require spatial and temporal data to conform to some regular structure (such as a 2D grid representing an image, or an ordered sequence representing regularly-spaced observations throughout time), in our setting all observations' spatial and temporal locations are represented by unconstrained continuous values. Consequently, the number of input observations and target queries in a problem instance is not fixed. Any problem instance may include any number of entities, each of which may have any number of representative samples included in the set of input observations.

Just as instances of such spatio-temporal problems consist of a dynamic number of samples in the input set, the number of queries associated with each problem instance is also not fixed. For example, the problem's goal may be to predict the state of each entity at some future time, or it may be to predict the state of some unobserved location in space and time from the provided input samples. In either case, the number of desired predictions and their relationship to the input samples is not fixed. This is in contrast to most deep models, which generally produce a constant-size output or produce outputs of varying length but over a fixed structure, such as a sequence.

We observe a problem instance via a set of input samples X which describes the state of the process' entities at certain locations and time points. Specifically, each sample $x \in X$ describes an entity uniquely identified by $Ent(x)$ located at position $Pos(x)$ at time $Time(x)$. The domain-specific information for each sample needed for inference is represented by the sample's feature vector $Feat(x)$. To represent the semantic meaning of the desired predictions, we also define a set of 'query targets' Q. Each domain-specific query $q \in Q$ describes the relationship between the desired prediction target and the samples represented in X. For example, in a domain in which the goal is to predict the future state of an entity in the process, each query q would specify the entity in question, $Ent(q)$, and the desired time point of the prediction, $Time(q)$. In a domain in which the goal is to predict the state of the process at some location rather than the prediction being associated with a specific entity, such as predicting the current weather conditions at an unobserved location, q would instead be defined by $Pos(q)$ and $Time(q)$.

An individual spatio-temporal problem instance is then defined by the tuple (X, Q) describing the input samples and desired queries. Models are trained on a labeled dataset $\mathcal{D} = \{\langle (X_i, Q_i), Q_i^* \rangle | i \in \{1, \ldots, N\}\}$, where each Q_i^* is the ground truths for the set of queries Q_i on the observed input data X_i. The models are evaluated on their ability to accurately predict the correct Q^* when provided with a problem instance (X, Q). This problem structure allows us to train models on spatio-temporal problems consisting of a dynamic number of input samples and a dynamic number of domain-defined prediction targets. Such models are expected to reason about the spatial and temporal relationships within the input samples X, as well as the domain-specific relationships between those encoded input samples and the desired query targets Q.

4 Benchmark Domains

We consider three spatio-temporal problem domains to support our evaluation: Starcraft II battle unit state prediction, weather nowcasting, and traffic forecasting. These problem domains demonstrate the models' ability to understand qualitatively different types of processes and queries. For example, observations from the Starcraft II domain describe the state of each individual unit present in the scene, whereas observations from the weather nowcasting and traffic forecasting domains represent point samples of the underlying process in question from fixed observation stations (that is, the atmospheric conditions recorded by weather stations and the traffic information recorded by road sensors). Queries in the Starcraft II and traffic forecasting domains task the model with predicting the future state of a specific entity described in the input observations, whereas the goal of the weather nowcasting problem is to use recent samples of nearby weather conditions to predict the current weather conditions at some unobserved location. Despite the significant differences in the underlying semantics of the input data and the structure of the queries, all problem instances are described as a spatio-temporal process as defined above. Specific information about each problem domain and how they are translated into a spatio-temporal problem instance is described below.

4.1 Starcraft II

The video game Starcraft II is a popular, challenging domain for evaluating machine learning models, most notably used by (Vinyals et al., 2019) as a challenge problem for reinforcement learning. It is interesting for our purposes primarily due to the dynamic nature of the military encounters represented by the game. These encounters may consist of just a couple of opposing units fighting one-on-one, or an entire battle with dozens of units on each side interacting with the others, making it a good platform to demonstrate how models can handle problems consisting of differing numbers of entities. Additionally, since the game engine reports the units' state at a high frequency, we can sample a subset of the recorded timesteps as input to the models to examine their ability to handle samples which are irregularly spaced in time.

The dataset is generated from a custom Starcraft II scenario in which two opposing armies fight each other on a featureless play field. Each scenario begins with a random number of three distinct unit types placed in random locations on the play field for both teams. The game then runs without any input, so the units perform their 'default' behavior of attacking any enemy units until they die or there are no enemies nearby. The scenario ends once all the units for one team have been defeated. Figure 1 shows an example of what a small encounter in this scenario looks like in-game.

Figure 1: Example SC2 scene

Name	Type	Size	Target?	Description
Owner	Onehot	1		Binary representation of the team the unit belongs to
Type	Onehot	3		Onehot encoding of the unit type (marine, zergling, zealot)
Heath	Real	1	✓	Current health value of the unit
Shields	Real	1	✓	Current shield value of the unit
Orientation	Onehot	7	✓	Direction the unit is facing
Position	Real	2	✓	Cartesian position of the unit on the game field

Table 2: Description of feature vector representation of each unit in the Starcraft domain. The variable is used as a prediction query target if the Target? column is checked.

We use the PySC2 API interface provided by DeepMind (Vinyals et al., 2017) to record the state of each unit at each timestep across 1000 runs of the scenario. See Table 2 for a complete description of the feature vector representation for each unit. Individual problem instances are derived from this dataset by first selecting one individual timestep from one of the scenarios. We then choose a subset of the recorded game states before that timestep to use as 'input frames', and choose a subset of the timesteps after the selected timestep to use as 'query times'. The models are then evaluated based on their ability to use the unit information provided from the input frames to correctly predict the state of each unit at each query time.

Specifically, when evaluating models on this domain we select the 'input frames' by randomly selecting up to five timestep frames within the previous ten before the timestep in question. All entities in each selected frame are included in the set of input samples X. We fixed the set of target 'query times' when training and evaluating models on this domain unless otherwise specified. Specifically, for problem instance at time t we task the model with predicting each unit's state at all of the timesteps $t+1$, $t+2$, $t+4$, and $t+7$ that exist. Each timestep is approximately half a second of in-game time, so the models are asked to predict the future state of each unit in the scene at 0.5, 1, 2, and 3.5 seconds into the future.

4.2 Weather Nowcasting

Predicting the current atmospheric conditions at a target location given a set of current or prior weather observations from nearby areas ('nowcasting') requires jointly reasoning about the spatial and temporal relationships between the target location of interest and the provided historic weather observations. We use

Name	Metadata?	Target?	Description
RELH		✓	Relative Humidity
TAIR		✓	Air Temperature
WSPD		✓	Average Wind Speed
WVEC			Vector Average Wind Speed
WDIR			Wind Direction (heading)
WDSD			Standard Dev. of Wind Direction
WSSD			Standard Dev. of Wind Speed
WMAX			Maximum Wind Speed
RAIN			Liquid precipitation since 00 UTC
PRES		✓	Station Pressure
SRAD			Solar Radiation
TA9M			Air Temperature at 9m
WS2M			Wind Speed at 2m
TS10			Sod Soil Temp at 10cm
TB10			Bare Soil Temp at 10cm
TS05			Sod Soil Temp at 5cm
TB05			Bare Soil Temp at 5cm
ELEV	✓		Elevation
LAT	✓		Latitude of station
LON	✓		Longitude of station
Soil Info	✓		A vector of length 18 describing soil properties

Table 3: Description of feature vector representation of each station in the weather domain. The variable is used as a prediction query target if the Target? column is checked. Metadata? is checked if the value is static and associated with the weather station in question.

a dataset of weather conditions recorded by a group of weather stations distributed throughout Oklahoma to derive such a weather nowcasting problem to evaluate the models.

The dataset consists of the atmospheric conditions and associated quality metrics recorded by each weather station in five-minute intervals throughout the entirety of 2008, as well as metadata describing each station's static properties (e.g. location, elevation, soil type). See Table 3 for a complete description of the feature vector representation for each station reading. Any samples with quality metrics that report their readings may not be accurate are removed from the dataset entirely. No imputation is performed to 'fill in' these missing data points, since our models are expected to be able to process such sparse data on their own. 90% of the stations are selected to be used as training data, while the remaining 10% are used for testing.

To derive a training problem instance from this dataset, we select a timestep t from the dataset and a subset of the training stations to use as 'target' stations. The set of input samples X consists of all weather station observations from the hour prior to the selected timestep t which do not come from a selected 'target' station. The set of queries Q describes the locations of each desired nowcasting prediction, which are the locations of all of the selected 'target' stations. The ground truth Q^* is then set to be the recorded observation by each target station at time t. Test problems are generated similarly, except that all the training stations' samples are included in the input to X, and the model is tasked with predicting the state of the held out test stations at the selected timestep t.

4.3 Traffic Prediction

Predicting traffic flow is a common problem to demonstrate the performance of spatio-temporal GNN models (such as by Yu et al. (2017) and Zhang et al. (2020)), as the signals of interest are highly periodic in time and exhibit significant spatial locality throughout the road network. We use the publicly available METR-LA dataset (Li et al., 2017), which consists of 206 sensors distributed through the LA highway system. The sensors report the average speed of the highway traffic at their location every five minutes. The dataset consists of all readings between between March 2012 and June 2012.

An individual problem instance is derived from this dataset by first selecting a timestep t in the dataset. The traffic network's sensor readings from the previous hour (that is, timesteps $t-11$ through t) are collected to use as the set of input samples X. The set of queries Q is defined to request the state of each sensor in the traffic network one hour in the future (that is, timestep $t+12$), and Q^* is set to be the observed traffic conditions at each sensor at that time. The models are evaluated on their ability to use the provided recent traffic data to predict the speed of traffic flow at each sensor's position in the road network at the selected target timestep. Note that the are no held out 'target sensors' that are hidden from the input data, unlike in the weather nowcasting domain. The goal of each problem instance is to use the historic data from all sensors to predict the future state of all sensors. Instead, we select the data from every tenth week from the dataset to use as test data, while the remaining 90% of the data is used for training.

Note that the input and query structures for this domain are fixed, unlike in the other domains. For Starcraft, the relative position of the units is always changing, and units may disappear at any time. For weather nowcasting, some stations' observations may not be present in each input timestep and the requested queries may change with each problem instance. In comparison, the input to each traffic prediction problem is a fixed size, as each traffic sensor is present at every timestep. Additionally, the queries always represent the same relationship– requesting the state of each sensor one hour in the future. This enables this specific problem formulation to be fully compatible with classic temporal models, such as RNNs. However, the spatial component of the problem is still incompatible with classic spatial models, as spatial gridding would still be necessary and CNNs would be unable to exploit the explicitly-defined graph structure of the road network's spatial layout.

5 Models

Graph networks are a class of artificial neural networks which operate on graph-structured data. These graphs consist of a set of nodes with associated feature vectors, and at least one set of edges describing the connectivity of the nodes. In our settings, edges also have associated feature vectors describing the relative information between the two nodes they connect.

Message-passing GNNs are a class of GNN that generally operate by using their input graph's adjacency information described by its edges and the attached nodes' feature values to calculate a latent 'neighborhood representation' for some (usually all) nodes in the graph. Note that the structure of the input graph in not constrained, so a node may have any number of neighbors and associated edges which contribute to its latent neighborhood representation. The resulting fixed-sized neighborhood representation is then combined with the node's feature vector to determine a new latent representation for each node which is meant to describe the 'context' from its immediate neighbors as it relates to the node in question itself. This process can be performed for any subset of nodes in the graph to calculate latent feature vector for those specific nodes. When performed on all nodes X present in the graph, the output of this encoding process is a set of latent feature vectors which correspond to all of the provided input nodes. By setting the graph's node features to be the corresponding latent feature vectors calculated by this application of the GNN, we get an updated latent graph with identical structure as the input but with latent feature vectors which hopefully represent useful information about each node's neighbors. This encoding process can then be repeated with any number of layers, similar to other DNN architectures, which effectively increases the 'range' at which information can propagate throughout the graph. We represent this process of using a GNN to calculate feature representations for a subset of nodes X in the graph G with $GNN(G, X)$.

In contrast, calculating the latent feature vectors for the query nodes Q is done with $GNN(G, Q)$. Note that the output of this GNN does not directly correspond with the nodes in the input graph G, since Q is a strict subset of the input nodes X. As a result, this process is not repeatable or layerable as the input encoding process is. Since the output of $GNN(G, Q)$ correspond to the queries Q, we finally pass each latent encoding of a query node through an MLP to calculate the final prediction for each query based on the input graph.

Expression	Meaning
X	Set of nodes
E	Set of edges
$G(X, E)$	Graph consisting of nodes X and edges E
e_{xy}	Edge connecting nodes x and y
$EdgeFeat(x, y)$	Feature value of e_{xy}

Table 4: Graph Encoding Notation

5.1 Graph Encoding

Since the models we examine operate exclusively on graphs, the process of deriving a graph representation of each problem instance is crucially important in enabling the models to effectively reason about the problem. We propose a simple technique for converting such spatio-temporal prediction problems into a multi-relational graph which captures the spatial and temporal relationships between relevant samples in the problem instance. See Table 4 for a description of the notation used in describing this process.

The core idea is to define three different sets of edges between the samples X described in the input spatio-temporal process instance:

- E_s to represent the spatial relationships between samples on each individual timestep, such as the distance vector between two interacting units in a Starcraft battle; and
- E_t to represent the temporal relationships between samples of each individual entity across all timesteps in which it is present, usually the difference in timestamps between ; and
- E_q to represent the domain-appropriate relationship between the problem's input samples and the specified query targets. For example, in the Starcraft domain in which the goal is to predict the future state of a specific unit, E_q is defined identically to E_t. In the weather domain, where our goal is to predict the weather conditions at a unobserved location at an observed timestep, E_q is defined identically to E_s.

Specifically, the set of temporal edges E_t and their values is defined as follows:

$$EdgeFeat_t(x, y) = Time(y) - Time(x) \tag{1}$$

$$E_t = \{\forall_{x \in X} \forall_{y \in X, x \neq y}\ e_{xy} \text{ if } Ent(x) = Ent(y)\} \tag{2}$$

The spatial edges are defined similarly. Due to the potentially large number of entities within a given timestep, spatial edges are only created between samples within some specified maximum interaction distance. Specifically, the set of spatial edges E_s and their values is defined as follows:

$$EdgeFeat_s(x, y) = Pos(y) - Pos(x) \tag{3}$$

$$E_s = \{\forall_{x \in X} \forall_{y \in X, x \neq y}\ e_{xy} \text{ if } |EdgeFeat_s(x, y)| < \Delta_{\max}\} \tag{4}$$

With these two disjoint sets of edges, we can realize two different graphs that share the same set of nodes but use the two sets of edges to represent both types of relationships: the spatial relationship graph $G_s = G(X, E_s)$, and the temporal relationship graph $G_t = G(X, E_t)$.

Encoding an input graph G by calculating a latent feature vector for each node X in the graph requires employing the spatial and temporal GNN on their respective graph representations, as well as MLP applied in

Algorithm 1

function SpatioTemporalEncoder(X, E_s, E_t, GNN_s, GNN_t, $NodeMLP$)
 $G_s \leftarrow G(X, E_s)$
 $Y \leftarrow GNN_s(G_s, X)$
 $G_t \leftarrow G(X, E_t)$
 $Z \leftarrow GNN_t(G_t, X)$
 $X' \leftarrow NodeMLP(X||Y||Z)$
 return X'
end function

Algorithm 2

function SpatioTemporalQuery(X, E_q, GNN_q, $QueryMLP$)
 $G_s \leftarrow G(X, E_q)$
 $Y \leftarrow GNN_q(G_s, X)$
 $P \leftarrow QueryMLP(Y)$
 return P
end function

parallel to each latent node representation to combine the output from each GNN into a latent representation for each node that combines information from its spatial and temporal neighborhoods.

The resultant latent encoding of the samples X' can then be fed into the next layer of spatial/temporal GNN pairs. See Algorithm 5.1 for a description of how one such layer is executed. Note that E_s and E_t are constant across all layers, and therefore only need to be determined once per problem.

Finally, we must use the resulting latent encoding of each sample to derive a latent encoding of each desired query target $q \in Q$. As before, we define a set of edges E_q that describes the relationship between the encoded samples X and the desired queries Q. The specific procedure for defining E_q is domain specific, as it depends on the type of relationship between the samples and the queries. Note that E_q must be a bijection between X and Q, representing the fact that the result of each query q is exclusively dependent on the latent encoding of the input samples, and not dependent on the other contents of Q itself. See Algorithm 5.1 for a description of how the query predictions P are derived from the graph from the encoding layers.

5.2 GraphConv

GraphConv (Kipf & Welling, 2016) is a simple message-passing GNN model designed to approximate a full spectral convolution of the input graph. This approach exclusively uses adjacency information from the graph's set of edges to determine how nearby nodes' features should be combined.

The GraphConv update rule is defined as follows:

$$X^{\ell+1} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}X^{\ell}W^{\ell}\right) \tag{5}$$

Expression	Meaning
X^{ℓ}	Input node features at layer ℓ
A	Adjacency matrix
D	Degree matrix
W^{ℓ}	Per-layer trainable parameter matrix for layer ℓ

Table 5: GraphConv notation

Expression	Meaning
X^ℓ	Node features at layer ℓ
a	Aggregation function – combines effects with their associated objects
ϕ_O	Object model – determines the future state of each object and its associated interactions
ϕ_R	Relational model – determines the effect of each interaction
m	Neighborhood/marshalling function – determines interactions and relative distances

Table 6: Interaction network notation

Where X is a $n \times f_1$ matrix of the nodes' feature values. $\tilde{A}$ is the $n \times n$ adjacency matrix A with added self-connections, that is $\tilde{A} = A + I_N$. $\tilde{D}$ is the $n \times n$ degree matrix, defined as $D_{ii} = \sum_j A_{ij}$. W^ℓ is the $n \times f_2$ weight matrix for layer ℓ.

Notably this approach completely ignores any features associated with the graph's edges. In our setting, these edges are assigned feature values that explicitly describe the spatial or temporal relationship between the two samples. Therefore, GraphConv cannot take advantage of this information directly. It must be inferred via some implicit pairwise comparison between a node and its neighbors' features. However, since the neighbors' features are collected purely via the operation multiplying the normalized adjacency matrix $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ by the nodes' feature values, there is no mechanism by which individual neighbors' features can be processed differently based on their relation to the 'central' node. Any such logic must therefore be expressed by the effects of the weight matrices W on the weighted combination of neighborhood and node features, which is unlikely to be an efficient or effective way to express those functions.

5.3 Interaction Networks

Interaction networks (Battaglia et al., 2016) are a general class of GNN designed to operate on problems involving many individual interacting entities. A single Interaction Network layer is described as follows (see Table 6 for notation):

$$X^{\ell+1} = \phi_O\left(a\left(G^\ell, \phi_R\left(m\left(G^\ell\right)\right)\right)\right) \tag{6}$$

In our setting, each function is defined as follows:

- m, the marshalling/neighborhood function, which operates on the provided graph $G(X, E)$. E is one of E_s, E_t, or E_q depending on the context in which the layer is being applied. The output of m represents each edge (or relation) in the graph with a tuple consisting of the source node's feature vector, the destination node's feature vector, and the edge feature itself describing the relative information between those two nodes. Specifically, $m(G(X, E)) = \{[x, y, EdgeFeat(x, y)] \mid \forall_{x \in X} \forall_{y \in X, y \neq x}, s.t. e_{xy} \in E\}$.
- ϕ_R, the relational model, which transforms each relation tuple from m into a fixed-length latent representation of that relation's 'effects'. Specifically, ϕ_R is implemented with a MLP that is applied in parallel to each relation tuple.
- $a(G(X, E), R)$, the aggregation function, performs two tasks: combining the variable number of latent relational effect vectors associated with each node into a single fixed-size 'neighborhood effect' vector, and combining this 'neighborhood effect' vector with the node's own latent feature representation. Specifically, for each node $x \in X$, $N_x = \sum_{r \in R_x} r$, where R_x is the set of relations in R which are associated with edges in E directed towards node x. The output of the aggregation function is then $a(G(X, E), R) = \{[x, N_x] \mid \forall_{x \in X}\}$, a set of pairs consisting of each node's feature representation and its aggregated relation latent vectors describing all its neighbors.

- ϕ_O, the object function, which transforms the pairs of node features and associated aggregated neighborhood features into a final latent representation for that node to be provided to the next layer. Specifically, ϕ_O is implemented with an MLP that is applied in parallel to each object-neighborhood pair.

At a high level, applying one layer of an interaction network performs the following steps:

- use the edge information provided by the input graph to determine all the neighboring nodes for each node in the graph;
- transform each neighbor relation into a latent feature vector based on the main node's value, the neighboring node's value, and the connecting edge's value;
- sum together all latent neighbor relations associated with each node into a single fixed-size neighborhood representation;
- finally, combine each node's latent feature representation with its neighborhood representation to produce an encoded latent feature representation for each node.

5.4 PointConv

PointConv (Wu et al., 2019) is a convolutional neural network designed to operate on point clouds rather than the image or voxel data that would be consumed by a traditional CNN. Whereas CNNs learn a set of discrete filters to apply over regularly-structured hyper-rectangles (usually 2D/3D images), PointConv instead learns a filter function represented by an MLP which transforms a distance vector between two points into the filter value for that location. The authors prove that PointConv is equivalent to traditional pixel-based image convolution

At a high level, PointConv performs the following steps:

- determine the 'neighborhood' of each point in the cloud by collecting all other points within a certain distance;
- calculate the relative distance between each point and each point in its neighborhood;
- transform each relative distance vector into a filter vector using a MLP;
- perform a matrix multiplication between the filter vector and its associated neighbor's feature vector;
- sum all resulting filtered neighbor vectors into a single fixed-size neighborhood representation;
- finally, combine each point's latent feature representation with its neighborhood representation to produce an encoded latent feature representation of each point.

Although it isn't presented as such in the paper, PointConv can also be described in terms of being a graph network. The only significant difference between the PointConv and Interaction Network algorithms is how the neighborhood feature vector is calculated by the relational model ϕ_R. While the interaction network simply calculates each neighbor representation with a MLP such that $r = MLP(x||y||EdgeFeat(x,y))$, PointConv calculates each neighbor representation as $r = F(y^{\intercal}W(EdgeFeat(x,y)))$, where W is a MLP that transforms an edge feature vector into a fixed-size filter vector, and F is a 'flattening' function that reshapes an $n \times m$ matrix into a $1 \times (n \times m)$ vector.

Notably, this filtering-based approach removes the neighbor vector's dependency on the feature value of the point (or node) for which the neighborhood is being calculated. As the filter value is only dependent on the relative distance between two samples, any pair of samples with the same relative distance will also have the same filter value returned by W.

5.5 PointConv with Attention

As described above, the filters calculated by PointConv do not depend on the feature value of the node x or its neighbor x'. While this allows us to explicitly calculate filter weights for each neighbor, it seems reasonable to expect that in some domains these neighbors should be filtered differently depending on both their feature descriptions.

We can extend PointConv to have such capabilities by applying a simple attention mechanism to the weight calculation, as described by Horn et al. (2019). Instead of calculating the weights exclusively with $W(EdgeFeat(x, y))$, we first calculate an attention vector $a = softmax(A(Feat(x)||Feat(y)))$, which uses an MLP A to calculate a vector of a small number of normalized scalar weights depending on the comparison between the node x and its neighbor y. We then multiply the filters by each value in the attention weights as follows: $r = F(x'^{\intercal}F(a^{\intercal}W(EdgeFeat(x, y))))$.

Note that now our neighbor representation r for a given node x and neighbor y is now dependent on the values of x, y, and the relative distance between them, $EdgeFeat(x, y)$. However, instead of accomplishing this by simply concatenating these values together and applying an MLP as Interaction Networks do, we explicitly learn a set of spatial filters that depend exclusively on the relative distance between the samples, and a set of weights that depend exclusively on the two samples' feature values.

6 Hyperparameter Selection and Training

The performance of deep learning models, is highly dependent on the hyperparameter configuration used for training. In some cases, the effect of thorough hyperparameter optimization on a model's observed performance may dominate the performance impact of that model's structural or procedural differences from the other models it is being compared to. As a result, when comparing different models' performance it is necessary to ensure that each model receives similar amounts of hyperparameter tuning effort to ensure that the comparison's results represent the differences in the models' inherent properties rather than the differences in their high-level training procedures. This can be problematic since many works presenting novel models do not make an attempt to quantify how much time or effort was put into determining the hyperparameters for training the model being evaluated. Future works that then use such reported results as comparison points cannot guarantee that the models being evaluated in the comparison are 'evenly matched' in terms of hyperparameter optimization effort. The widespread assumption is then that any presented results are from a hyperparameter configuration which 'maximizes' the performance of the model in question. This assumption incentivizes authors to commit significant time and resources to optimizing hyperparameters to improve their proposed model's performance without necessarily recording or describing that optimization process, since it is generally assumed that the comparison models (whose performance frequently must be matched for the work to appear relevant) also had significant time and resources committed to maximizing their performance.

We intentionally take a different approach for our 'stake-free' setting by defining a single hyperparameter selection procedure to be applied to each instantiation of each model in the evaluation. Explicitly making the hyperparameter selection process part of the evaluation procedure allows us to guarantee that each model received a fair amount of hyperparameter tuning 'effort' since we can demonstrate that the selection process and resources used were the same for all models. The simplest common approach for hyperparameter optimization is grid search or random search (Bergstra & Bengio, 2012), in which every hyperparameter is enumerated and regular or random samples drawn from the resulting hyperparameter configuration space are evaluated with an auxiliary training procedure. The hyperparameter configuration with the best empirical performance demonstrated by the auxiliary training procedure is then selected to be the representative hyperparameter setting for the model in question. This approach can be very expensive due to the massive number of training runs that must be executed to ensure good coverage of the hyperparameter configuration space. Since we need to perform the hyperparameter selection for each size, for each model, for each domain, performing a complete grid search is infeasible.

To determine a reasonable hyperparameter settings within a reasonable amount of time, we fix most training hyperparameter values to commonly-used defaults, and focus exclusively on determining an appropriate

learning rate schedule for each model instantiation, as the learning rate has been shown to generally be the most impactful hyperparameter in determining model performance (Smith, 2017). Specifically, for all experiments we use the ADAM optimizer (Kingma & Ba, 2014) as implemented by PyTorch (Paszke et al., 2019), using its default parameters of a weight decay of 0, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. We use a cyclic cosine annealing learning rate schedule as described by Loshchilov & Hutter (2016), with T_{mult} set to 2 and T_0 set to $\frac{1}{7}$th of the total number of epochs to ensure that the schedule can complete three periods during each training run. The cosine annealing schedule's minimum and maximum learning rates are determined experimentally for each model instantiation using a modified learning rate test described below.

6.1 Checkpoint Learning Rate Test

Smith (2017) describes a procedure for quickly determining a range of usable learning rates for training any given network. The key idea is to perform an auxiliary training run on a randomly initialized network using the intended optimizer, while exponentially interpolating the optimizer's learning rate from a low value known to have no meaningful impact on the network's performance to a high value known to cause the network to 'explode' and lose performance. This is in opposition to the usual approach for training a deep network, in which we would start with an appropriately large learning rate to allow the optimizer to find a promising parameter region, then gradually lower the learning rate to 'refine' its behavior with smaller parameter updates. Instead, the goal of a learning rate test is to quickly 'scan' through a wide range of plausible learning rates and examine the resulting effect on the performance behavior on the model being tested. The expectation is that there will be three regions which can be identified from these results:

- a 'too cold' region in which the learning rate is too low to impact the model's parameters, and the observed training loss 'flatlines';
- a 'just right' region in which the learning rate is effective and enables the model to improve, causing the observed training loss to fall;
- and a 'too hot' region in which the high learning rate causes the parameter updates to 'explode' the model, causing the observed training loss to quickly increase.

One can then assume that any learning rate from the 'just right' region is appropriate to use when training the final model.

This simple learning rate test procedure ignores some crucial facts about training deep networks. Specifically, the concept of a learning rate schedule is validated by the observed evidence that the most effective learning rate for an optimizer can change significantly over time as the model moves into effectively different regions of its parameter space. However, the learning rate test seems to make the opposite assumption– that the most effective learning rate does *not* change even as we update the model's parameters. By constantly updating the model's parameters as the learning rate is varied throughout the test, the test is essentially modifying its underlying problem domain while simultaneously searching for a solution within it. This calls into question the general utility of the results of such a test.

Fortunately this shortcoming is easy to address. We propose a simple modification to the learning rate test, the 'checkpoint learning rate test', in which the network's parameters are reset to a fixed state after each reported training loss. This change significantly reduces the 'domain shift' effect imparted by constantly updating a single model throughout the test. Instead, by constantly resetting to a checkpoint the observed model's parameters are never able to get too 'far' from the checkpoint model's parameters, increasing the chance that the test's results represent the expected behavior of the checkpoint model. Algorithm 3 describes our proposed algorithm for performing the checkpoint learning rate test, while Algorithm 5 describes the procedure used to determine the appropriate minimum and maximum learning rate from the output of a learning rate test.

Adding a checkpoint addresses the learning rate test's 'domain shift' problem, but still does not change the fact that effective learning rates generally change over time as a deep model is trained. If the checkpointing learning test is run with a randomly initialized parameter configuration as the checkpoint, then its results

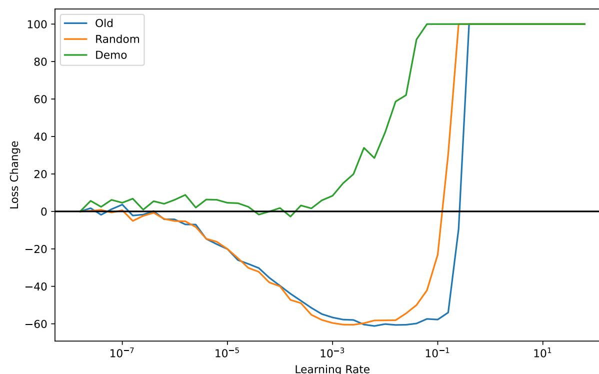


Figure 2: The results of three styles of learning rate test applied to the same network architecture. Old is the classic learning rate test, Random is the random checkpoint test, and Demo is the 'poorly trained' checkpoint test. Note the clear area of improvement for the Old and Random tests, while Demo has a much less clear area of improvement and 'explodes' at a much lower learning rate.

will represent the training characteristics of a randomly initialized network, such as one at the very beginning of a training run. However, the learning rate test is intended to determine an appropriate learning rate range to use throughout the entire training procedure, not just the very beginning. In fact, since networks tend to become more sensitive to the learning rate being used as they are trained, exclusively using the network's random initial conditions to determine acceptable learning rates for the entire training procedure may be of limited value.

To ensure the results of the learning rate test are useful for training across more of the training phase, we first apply a common auxiliary training procedure to produce a 'poorly-trained' network. Using this 'poorly-trained' network as the checkpoint results in an estimated learning rate range which is much more appropriate for the entire training process, since its parameter configuration is much closer to the state the network will be in during training compared to the initial random state. We found that networks trained with the learning rate range derived from this 'poorly trained' network were much more likely to converge in some domains than networks trained with the learning rate range derived from the random checkpoint or classic learning rate test. Figure 2 demonstrates this by showing the results of a learning rate test with three different approaches: the classic learning rate test, the test with random checkpointing, and the test with 'poorly trained' checkpointing.

This demonstrates that the effective learning rate range for a model changes as the model's parameters are updated during training. From this evidence, seems reasonable to then conclude that a single learning rate test is unlikely to be able to determine an appropriate learning rate for the entirety of the training procedure. However, since our modified learning rate test does not suffer from the same failures as the original learning rate test, we employ it to determine the learning rate ranges for every model instantiation across all domains. Further examining and developing this style of hyperparameter test is left to future works.

7 Results

In this section we present the results of our empirical investigation. We first, present an overall summary of the main observations. Next we provide additional details for each of the benchmark problems.

7.1 Overall Performance

Table 7 shows the loss statistics for all network architectures and sizes on all domains. In it, and all other figures in this paper, the PointConv-based network is abbreviated as 'PC', PointConv with attention is abbreviated as 'PCA', GraphConv is abbreviated as 'GC', and Interaction Network is abbreviated is 'Int'. Rather than report mean test loss, to avoid sensitivity to outlier predictions, we report the 25%, 50%, and

Algorithm 3 Checkpoint LR Test

```
function CheckpointLRTest(net, θ, dataset)
    SetNetworkParams(net , θ)
    batchCount ← 0
    losses ← empty list
    nextReport ← reportEvery
    for batch in dataset do
        batchCount ← batchCount + 1
        pred ← Predict(net, batch)
        loss ← CalculateLoss(pred, batch)
        lr ← LRSchedule(batchCount)
        UpdateParams(net, loss, lr)
        Append loss to losses
        if batchCount ≥ nextReport then
            nextReport ← nextReport + reportEvery
            SetNetworkParams(net, θ)
            RecordLoss(lr, mean(losses))
            losses ← empty list
        end if
    end for
end function
```

Algorithm 5 Learing Rate Calculation

```
function CalculateLRRange(lrs, losses, margin)
    baseline ← losses[0]
    highThresh ← baseline × margin
    highIdx ← min i such that losses[i] > highThresh
    lrs ← lrs[0:highIdx]
    losses ← losses[0:highIdx]
    lowScores ← [Score(i, lrs, losses, baseline) for i in 0..length(losses)]
    lowIdx ← argmax_i lowScores[i]
    lowLR ← lrs[lowIdx]
    highLR ← lrs[highIdx]
    return (lowLR, highLR)
end function
function Score(idx, lrs, losses, threshold)
    target ← [loss < threshold for loss in losses]
    pred ← [i ≥ idx for i in 0..length(losses)]
    score ← F1Score(pred, target)
    return score
end function
```

Domain	SC2					Weather					Traffic				
Network	P_{25}	P_{50}	P_{75}	Mean	%Fail	P_{25}	P_{50}	P_{75}	Mean	%Fail	P_{25}	P_{50}	P_{75}	Mean	%Fail
GC-Small	1.37	5.85	14.76	9.48	0.0	**2.38**	5.85	13.05	10.07	0.2	0.77	3.07	14.06	45.92	**0.0**
Int-Small	**0.20**	**2.43**	**9.61**	**6.50**	0.0	4.89	9.64	18.31	14.09	0.4	0.70	2.73	11.03	43.95	<0.1
PC-Small	0.63	4.18	12.15	7.97	0.0	2.77	**5.69**	**11.22**	**9.00**	**<0.1**	0.77	2.81	11.24	45.17	<0.1
PCA-Small	0.33	3.04	10.64	7.10	0.0	21.48	34.05	53.13	41.76	0.4	**0.51**	**2.17**	**10.69**	**42.26**	0.2
GC-Med	0.84	4.81	13.15	8.51	0.0	1.26	2.63	5.49	4.72	<0.1	0.83	3.22	13.78	45.79	<0.1
Int-Med	**0.14**	**2.01**	**8.79**	**6.06**	0.0	0.61	1.42	3.40	3.16	**0.0**	0.68	2.86	13.69	45.39	**0.0**
PC-Med	0.36	3.26	10.78	7.17	0.0	**0.43**	**0.89**	**1.98**	**1.94**	**0.0**	**0.40**	**1.88**	**9.68**	**36.44**	0.9
PCA-Med	0.25	2.68	10.02	6.73	0.0	20.67	34.40	53.49	41.74	0.3	0.41	1.96	9.89	37.57	0.8
GC-Large	0.74	4.52	12.71	8.26	<0.1	5.43	9.11	17.06	20.98	8.7	**0.39**	2.08	13.55	46.75	**0.0**
Int-Large	**0.14**	**1.85**	**8.40**	**5.85**	**0.0**	**0.91**	**1.91**	**3.95**	**3.54**	**0.0**	1.76	5.63	19.08	47.16	**0.0**
PC-Large	0.27	2.81	10.10	6.79	**0.0**	1.32	2.79	5.85	4.84	0.8	0.42	**1.97**	**9.86**	**38.44**	0.8
PCA-Large	0.21	2.45	9.48	6.44	**0.0**	7.58	14.06	24.30	18.11	0.2	0.54	2.38	10.45	43.05	0.4

Table 7: Loss Table

75% percentiles of loss values among all individual predictions each model made on each domain, in columns labeled P_{25}, P_{50}, and P_{75}. We then identify 'bad' predictions by identifying all predictions with a loss 100x higher than the 75% percentile of the loss values. These predictions are labeled as failures, and the column labeled %Fail shows the percentage of all predictions for that model that were identified as bad. Finally, we report the mean of all losses not identified as bad in the column labeled Mean.

GraphConv is clearly not competitive with the edge-aware GNN models overall. Its performance is especially poor on the Starcraft 2 domain, in which being able to differentiate near and far units is critical to predicting what action each unit will take. This supports the idea that edge-aware GNN models should be applied to graph problems with highly dynamic structures, whereas GraphConv may be better suited for graphs which always have the same or similar structures.

Accordingly, Interaction Networks dominate the Starcraft 2 domain, suggesting that their black box MLP relational model is significantly more performant that the decomposed versions used by PointConv. The Interaction network based models seemed to perform best at their largest size, dominating both the Starcraft 2 and Weather Nowcasting domains among large models. This may suggest that the black box relational model approach is more efficient at higher parameter counts than the biased convolution-style approach employed by PointConv, but more research would be necessary to investigate this effect in detail.

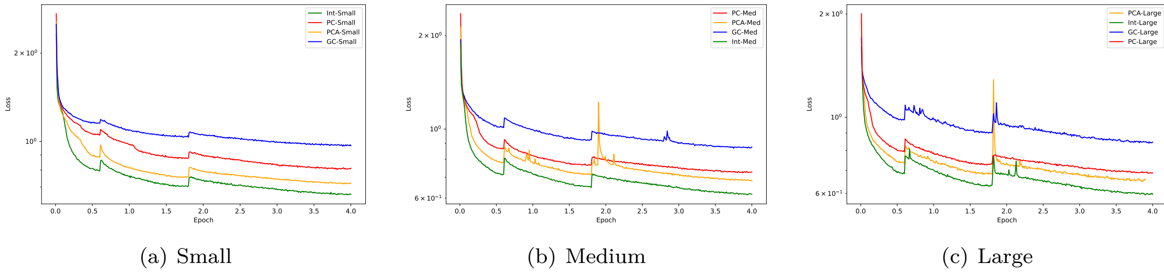
PointConv with Attention significantly outperforms PointConv in the Starcraft 2 domain, but appears to have completely failed on the weather problem. With a 25th percentile loss more than 10x higher than the other models for the small and medium sizes without any extreme prediction errors resulting in failures, it's clear that almost all of the PointConvAttention's predictions in this domain are useless. It's unclear why the addition of an attention mechanism may have caused this, considering that the extremely similar PointConv model performs well on the same domain.

The PointConv models seem to perform their best in the Traffic domain and the Weather domain (other than the large size). This may be related to the fact that these domains both have static graph structures, which may favor PointConv's approach of explicitly learning spatial filters, whereas in the Starcraft 2 domain which has extremely dynamic graph structures and entity interactions the Interaction Networks, with their explicit pairwise comparisons between samples, are much more performant.

Overall the results confirm our suspicion that when trained with equal amounts of hyperparameter tuning and training effort, no one GNN model dominates the others across all domains. Instead, the most important thing is to select the model with the inductive bias that best matches the properties of the problem instances that the model will be tasked with. The evaluation results suggest that PointConv-style, convolution-inspried architectures may be preferable when the graph structure is fixed, while the more entity-focused Interaction Networks may be a better fit for problems with highly dynamic graph structures.

7.2 Starcraft II

Training Curves. Figure 3 shows the average training loss across all training runs for each architecture instantiation on the Starcraft 2 dataset. The models' ordering is identical to that observed when evaluating



(a) Small (b) Medium (c) Large

Figure 3: Average training loss plots for all instances of each model trained on the SC2 dataset.

them on the test data, suggesting no overfitting is occurring. PointConv with Attention is noticeably unstable during training, however. This may be due to the learning rate test failing for this architecture and calculating too high of an initial learning rate, since the loss spikes seem to occur shortly after the cyclic learning rate schedule returns to its highest learning rates.

7.2.1 Spatial Filter Visualization

Figure 4 shows several of the continuous spatial filters learned by a PointConv model trained on the Starcraft 2 dataset. The filters seem to focus on the area in a tight radius around the unit in question with the further positions generally being constant, especially in the early layers where sufficiently far units will not have any impact on the unit in question's future state. This shows that the network is learning meaningful filters to gather information about the nearby units. If the learned filters were uninformative, the filters would appear to be a random projection– instead, we can clearly identify shapes that we know are directly relevant to the underlying spatio-temporal process' behavior. Note that the PointConv architectures are the only ones capable of producing such visualizations, since GraphConv and Interaction Networks do not learn a explicit weight function.

Attention. As with PointConv's learned weights, PointConv with Attention's attention mechanism can also be demonstrated by visualizing the attention weights assigned to each unit in a neighborhood. We select a random timestep and three random units, then highlight all the other units in its neighborhood with a color corresponding to their attention weights. As our attention mechanism learns three different weights, each of which ranges between 0 and 1, we just display each triple of attention weights as its corresponding RGB color.

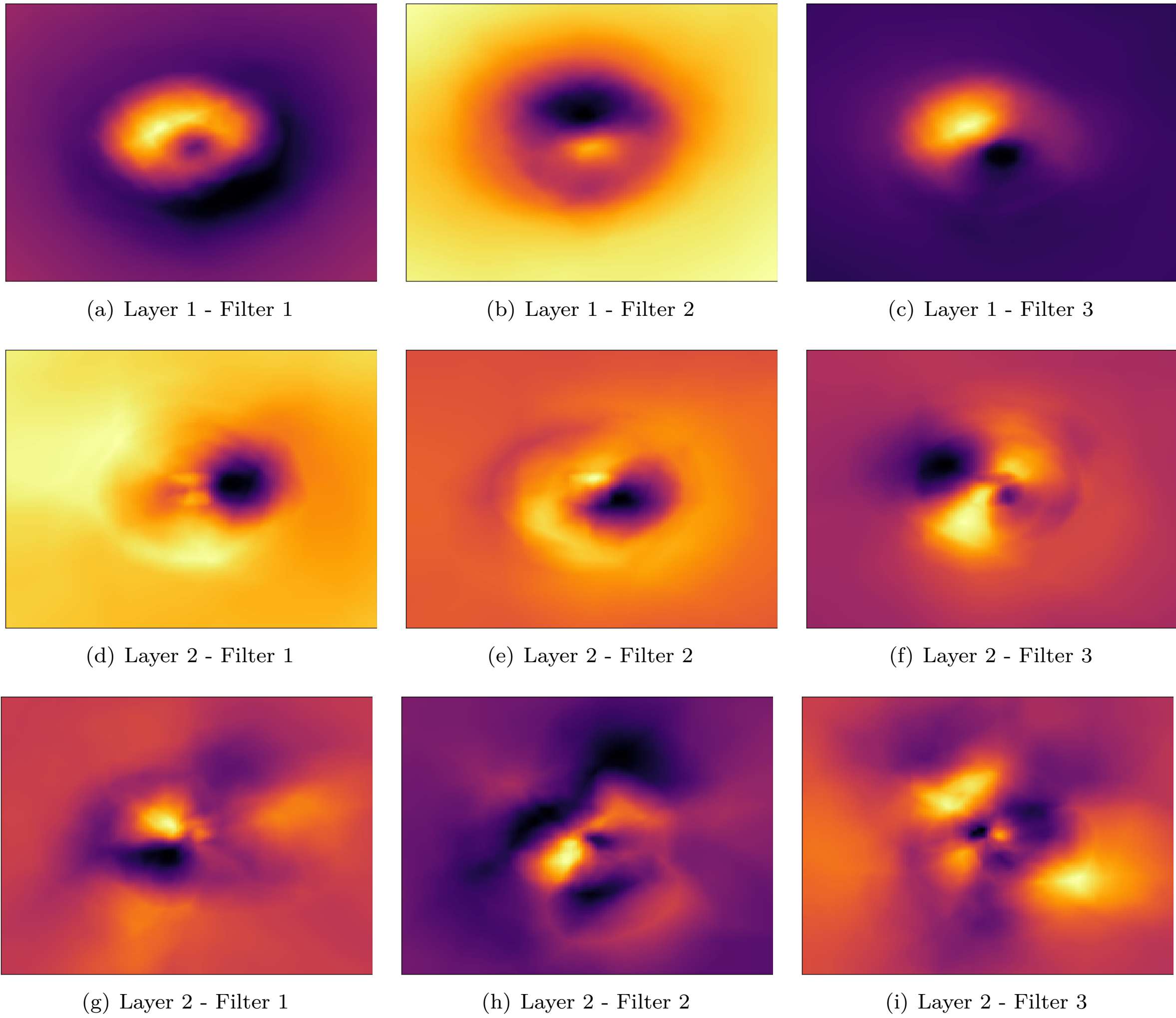
Figure 5 demonstrates that PointConv with Attention learns meaningful domain-specific attention weights. Specifically, the nearby enemy units clearly receive a lot of attention, while more distant units and friendly units tend to be less active. This lines up perfectly with what we expect, as the default behavior of all units in Starcraft 2 is to run towards and attack any enemy unit within close range. The significant performance gain over vanilla PointConv as well as the attention mechanism's clear understanding of the dynamics of the domain clearly demonstrate that this architecture is a good fit for the Starcraft 2 domain.

7.2.2 Query Timestep Distribution

One of the most notable features about our spatio-temporal problem setting is that the input and query target timesteps are not fixed. Rather, the model's capabilities and performance depend on the distribution of inputs and queries it was trained on.

We examine the 'out-of-bounds' behavior of these models by evaluating them on a modified Starcraft II dataset in which they must predict the future state of timestep offsets that were not present in the training dataset. Specifically, we show their average prediction loss for each query target timestep from $T + 1$ to $T + 12$, despite the fact that the networks were only trained on targets $T + [1, 2, 4, 7]$.

Additionally, we train these models in two addition settings: 'TwoPred', in which they are trained on a modified dataset where the query target offsets are set to $T + [2, 7]$; and 'OnePred', in which the query target



(a) Layer 1 - Filter 1 (b) Layer 1 - Filter 2 (c) Layer 1 - Filter 3

(d) Layer 2 - Filter 1 (e) Layer 2 - Filter 2 (f) Layer 2 - Filter 3

(g) Layer 2 - Filter 1 (h) Layer 2 - Filter 2 (i) Layer 2 - Filter 3

Figure 4: Filters learned by a PointConv model trained on the Starcraft 2 prediction problem.

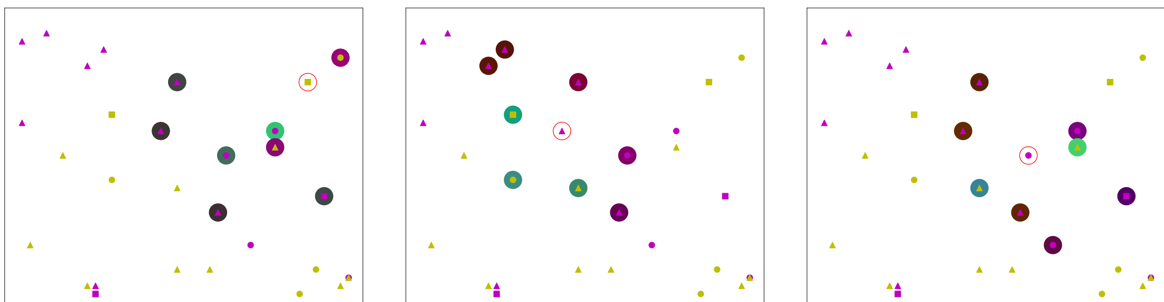


Figure 5: PointConv Attention visualization. The 'main' unit is circled in pink. Each unit in the neighborhood is highlighted with a solid circle colored according to its attention weights. Each unit icon's shape indicates its unit type, while its color indicates its team.

offset is only $T + 7$. Figure 6 shows a bar chart demonstrating the loss for each model trained on each query distribution on these out-of-bounds query target timesteps.

GraphConv. The GraphConv model appears to be least affected by being trained or evaluated on differing query timesteps. Despite no GraphConv having never observed any query timesteps above $T + 7$, they do not exhibit any extreme prediction errors that are indicative of overfitting. However, the performance of GraphConv models seems to increase when trained on smaller number of query target timestep offsets. This is not desirable behavior, since we would hope that giving the models more information on how the units change over time would lead to a better model of the units' behavior. Instead, the 'OnePred' model has clearly specialized to only focus on the specific timestep it was trained on ($T + 7$), and performs slightly worse on any other timestep. One would expect that 'TwoPred' should out-perform 'OnePred' on timestep $T + 2$, as 'OnePred' never been trained on targets at $T + 2$. Instead, 'TwoPred' preforms worse at almost all timesteps. Similarly, the 'default' setting (training on $T + [1, 2, 4, 7]$) performs worse than its more restricted counterparts despite effectively receiving more training data.

This suggests that GraphConv is not able to effectively distinguish between the different query targets it is trying to predict. This aligns with our intuition that GraphConv is significantly hampered by being unable to explicitly exploit edge features in a graph. Instead, the GraphConv model seems to have fallen back to underfitting behavior, in which it is unable to distinguish between the different query timestep offsets it must predict. In this mode of operation the loss would be expected to decrease as the diversity between the training targets decreases, which is exactly what we observe. This result shows that GraphConv is not an effective model to employ in this setting, and likely is ineffective on most graph problems with meaningful edge features.

Interaction Networks. The interaction networks' performance is nearly indistinguishable across all timesteps when trained on the default and 'TwoPred' settings. This suggests that both networks are able to learn similar, meaningful unit state transition models despite being trained on different query target timesteps.

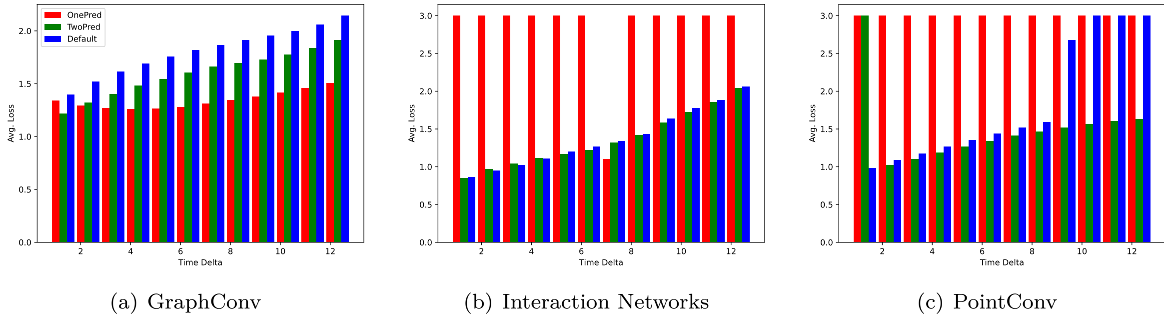
However, the interaction network trained on the 'OnePred' setting has clearly overfit. It is effectively useless at predicting unit states at any timestep other than $T + 7$, and only performs slightly better than the other models at its one training target $T+7$ itself. Note that adding just one additional query target during training (that is, query target offsets $T + [2, 7]$ instead of just $T + 7$) causes the model to go from overfitting on a single timestep to learning a general transition model which performs reasonably well across all timesteps. This seems to validate that applying graph models to this kind of graph realization of a spatio-temporal process is an effective way to enable the models to understand the process' dynamics in general.

PointConv. The PointConv networks' behavior is similar to the interaction networks' behavior, but PointConv appears to be more prone to overfitting behavior on unseen timesteps. The model trained on the default setting performs well up until $T + 8$, the first timestep it has never encountered during training, after which the prediction error rapidly increases. Alternatively, the model trained on the 'TwoPred' setting (that is, only on $T + [2, 7]$) exhibits overfitting behavior at the very earliest timestep it's never seen before ($T + 1$), but appears to do a better job at making coherent predictions for more distant targets compared to the default setting. Finally, the model trained on the 'OnePred' setting has completely failed to train and produces erroneous predictions at all timesteps on the test problem instances despite achieving reasonable performance during training.

Both these behaviors – a discrepancy between training and test performance, and training on more data resulting in worse general-case performance – are indicators of overfitting. This suggests that PointConv's learned filters, that it must rely on to reason about relationship between queries, is not as effective at generalization as the interaction network approach of replacing the filtering mechanism with a black-box MLP.

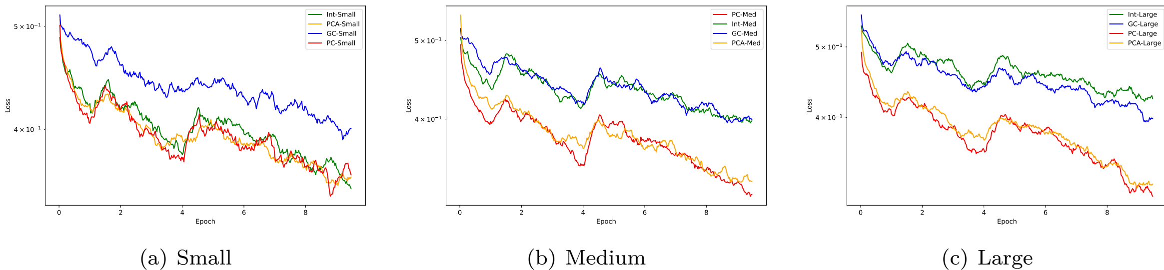
7.3 Traffic Prediction

Training Curves. Figure 7 shows the average training loss across all training runs for each architecture instantiation on the traffic prediction dataset. Based on the lack of clear convergence among any of the



(a) GraphConv (b) Interaction Networks (c) PointConv

Figure 6: Plots showing the average prediction loss for each model type for each timestep offset between T+1 and T+12.



(a) Small (b) Medium (c) Large

Figure 7: Average training loss plots for all instances of each model trained on the traffic prediction dataset.

models trained, it seems likely that the models could be significantly further improved with additional training time.

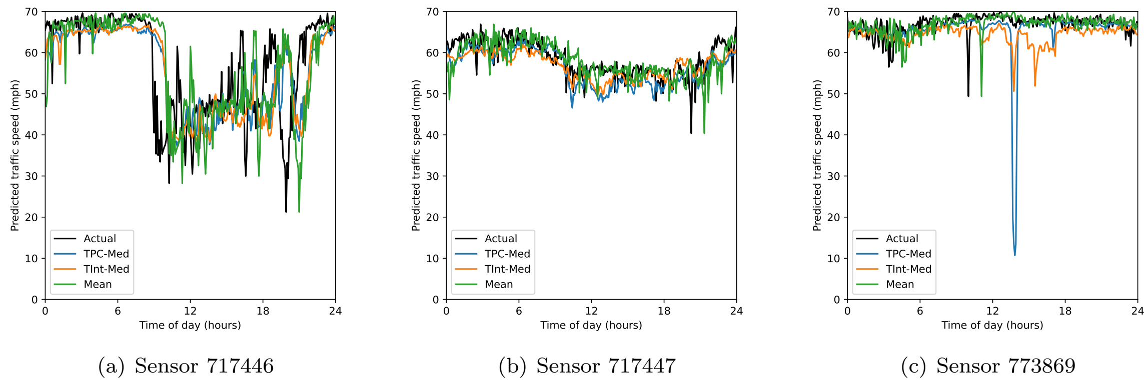
Baseline Comparison. To evaluate the models' usefulness, we compare its average performance to a simple baseline. Specifically, we use the baseline which simply predicts that the traffic speed at any given sensor in one hour will be the same as the sensor's current recorded speed. Interestingly, while this baseline model significantly outperforms all deep models in its first, second, and third quantile performance, the deep models significantly outperform the dumb baseline on average.

Individual Sensor Predictions. To gain a better understanding of the deep models' behavior and performance, we plot the predictions of each medium-sized network for a single sensor throughout one entire day. By comparing the predicted traffic speed to the actual signal, we may be able to gain some insight into how or why each model is failing.

Figure 8 shows an entire day's worth of predictions from each model for three different sensors. The one-hour delay from the baseline model is clearly visible, causing it to always miss quick changes in traffic speed by an hour. However, the deep models aren't much better at this. For example, in Figure 8(a), while they seem to start predicting a declining speed well before the baseline is able to, indicating they're using information from surrounding sensors to detect the oncoming traffic jam before it can be observed at the sensor, the prediction is still far off from the actual speed at that time and the model clearly 'follows' the baseline model's plunge in predicted speed as soon as it has access to sensor readings showing that traffic is stopped now. Additionally, the deep models seem to constantly predict slightly too low of a speed, as if they are hedging their bets expecting a traffic jam to materialize. These two observations may be a demonstration of the commonly observed phenomenon in which graph networks tend to 'underfit' or produce decent but clearly biased results in this style of prediction problem. It's unclear whether this issue may disappear with more training or if it is inherent to the GNN architecture itself.

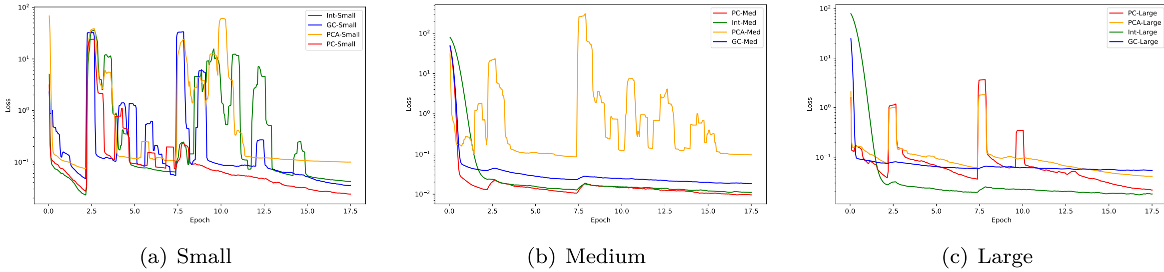
Network	P_{25}	P_{50}	P_{75}	Mean
Nearest	0.75	2.10	5.88	7.59
GC-Small	1.78	3.55	7.60	**7.54**
Int-Small	1.69	3.35	6.73	**7.15**
PC-Small	1.77	3.40	6.79	**7.26**
PCA-Small	1.44	2.99	6.62	**7.02**
GC-Med	1.85	3.64	7.52	**7.57**
Int-Med	1.67	3.43	7.50	**7.43**
PC-Med	1.28	2.77	6.30	**6.90**
PCA-Med	1.30	2.83	6.37	**6.92**
GC-Large	1.26	2.92	7.46	**7.31**
Int-Large	2.69	4.81	8.85	8.45
PC-Large	1.31	2.84	6.36	**6.99**
PCA-Large	1.50	3.13	6.55	**7.14**

Table 8: Table showing the 25%, 50%, 75% percentile, and mean prediction error for each model in MPH. Bolded entries indicate the model beat the baseline's performance.



(a) Sensor 717446 (b) Sensor 717447 (c) Sensor 773869

Figure 8: Each model's predicted traffic speed throughout an entire day for individual sensors. Black line is the target signal.



(a) Small (b) Medium (c) Large

Figure 9: Average training loss plots for all instances of each model trained on the weather dataset.

7.4 Weather Nowcasting

Training Curves. Figure 9 shows the average training loss across all training runs for each architecture instantiation on the weather prediction dataset. However, these training plots clearly indicate there's significant training instability. This suggests a failure in the learning rate test's ability to consistently determine appropriate hyperparameters for training. Clearly the learning rate is significantly too high for much of the training– but once it lowers towards the end, the networks' behavior seem to recover.

Station Dropout. In domains such as Starcraft II each observation directly represents an individual unit's state. Adding or removing a unit and its corresponding observations from a Starcraft II prediction problem will significantly change the expected dynamics of the process, since the units' behavior is highly dependent on the presence of other units in the scene. In contrast, in the weather problem domain each observation from a station is a point sample of the underlying continuous process of interest, the atmospheric conditions within the area. Since we cannot directly observe the entire the process, models must instead infer its overall state from these individual point samples.

Ensembling is a common technique to increase prediction performance by aggregating several predictions from multiple models instead of relying on a single model's prediction. Usually this is accomplished by training multiple models and averaging their predictions together. The graph structure of our problem and our understanding of the semantics of the domain suggest a different approach. We can instead provide a single model with several augmented problem instances whose input data has been modified while the queries are fixed. This single model's predictions on different realizations of the original input data can then be averaged together to perform 'self-ensembling'.

In this domain, we can augment a problem instance by randomly removing a percentage of the input samples while keeping queries unmodified. We demonstrate this self-ensembling approach by having each model produce prediction for five augmented problems derived from the original problem instance, and average the model's predictions together to produce the final prediction. Table 9 shows the mean and median performance of each model architecture as the percentage of dropped input samples is set to values between 0% (no augmentation) and 50%. We also show the performance of each architecture when trained with a 20% drop rate (models in the table postfixed with Drop20).

Among all models, increasing the amount of test-time station dropout generally decreases the model's prediction performance. However, the interaction network models trained in the 20% dropout setting significantly outperform the interaction network model trained on the un-augmented dataset regardless of the amount of test-time dropout. PointConv trained with dropout performs slightly worse than its default setting at 0% test-time dropout, but as the amount of test-time dropout increases it demonstrates a significant advantage over the model trained in the default setting. This is most obvious at 50% dropout where the default PointConv appears to be exhibiting overfitting behavior and producing erroneous predictions, while the model trained with dropout is able to outperform every other model.

These results show how applying appropriate graph data augmentation during training and evaluation effects the performance of these graph models. The models' overall performance did not significantly improve when only test-time data augmentation was applied. However, training with this data augmentation does not

Test-time Dropout	0%		10%		20%		50%	
Network	P_{50}	Mean	P_{50}	Mean	P_{50}	Mean	P_{50}	Mean
GC-Med	2.57	4.64	2.55	4.59	2.72	4.87	3.47	7.54
Int-Med	1.46	3.23	1.85	3.76	2.30	4.45	5.61	9.13
PC-Med	0.78	1.89	1.21	2.56	1.99	3.62	11.02	24.82
GC-Med-Drop20	2.93	5.18	2.72	4.92	2.68	5.06	3.30	8.51
Int-Med-Drop20	1.22	2.90	1.40	3.09	1.61	3.37	2.40	4.61
PC-Med-Drop20	0.90	1.97	1.03	2.21	1.21	2.54	2.17	4.10

Table 9: Table demonstrating the change in each model's test loss as station dropout is increased. Columns show the median and mean loss as the test-time station dropout is raised from 0% to 50%. Models with Drop20 appended to their name were trained with 20% station dropout.

significantly negatively impact the models' best-case performance while greatly increasing their ability to make effective predictions as the characteristics of the input data changes.

8 Evaluation Platform

One of the main goals of this work is to provide an approachable software platform to allow others to fully reproduce the experiments run for this paper, or modify and extend them if desired. In contrast to some other research codebases, we define general implementations of each model type and problem domain. Our training engine loads human-readable configuration files which describe the desired configurations for the model, problem, and training hyperparameters for the experiment it represents. This approach of separating the model and problem implementations from the specification of each experiment or training run reduces the barriers to running large sets of diverse experiments, such as those examined in this paper. Specifically, we have published a codebase which includes:

- Scripts to fetch each dataset used in the evaluation;
- Implementations of dataset loaders which derive graph representations of spatio-temporal problems from the raw datasets;
- Implementations of GraphConv, PointConv, Interaction Networks, and all of their components;
- Experiment definition files which configure the datasets and networks to train all the models used in this evaluation;
- Scripts to collect the results from training and produce all the plots and tables included in this paper;
- Instructive documentation on how to set up and run the code.

The repository for this project can be found at [TODO: removed for anonymous review].

9 Conclusion

We proposed a simple procedure to encode a spatio-temporal problem using a graph structure, including describing dynamic domain-specific queries. This graph approach enables us to describe a variety of problem types in one common format. Additionally, the graph structure allows for trivial data augmentation (when supported by the domain's semantics) and the query structure allows one to define multiple simultaneous prediction targets which force the models to learn the underlying process dynamics rather than a single relationship between the input data and the desired prediction.

We extended the 'learning rate test' concept, showing how it can be modified to more robustly identify an effective learning rate region for each individual model instantiation. This effort allowed us to determine

an effective training hyperparameter region for each individual model instantiation on each problem type, which is a process that would have otherwise been prohibitively expensive. However, we find that while this approach was reasonably effective in practice it still has significant shortcomings that caused undesirable training behavior in some cases.

Of the graph models we evaluated, we found that GraphConv consistently performed the worst, seemingly underfitting to most problems as it was unable to demonstrate it had learned the dynamics of the problems it was trained on. This is almost certainly owing to the fact that our graph realization of spatio-temporal problems stores relative information about neighboring nodes in edge features connecting them. Since GraphConv cannot exploit these edge features, it cannot take advantage of this useful bias.

Interaction Networks and PointConv seem to make meaningful predictions on all domains. This demonstrates that they are able to learn the dynamics of the problem they are trained on in some useful way. As PointConv learns an explicit weight function to filter neighboring nodes in the graph, we can visualize and interpret this weight function to validate that it is appropriate for the domain. Interaction networks are largely composed of black-box functions, and lack this interpretability. However, they perform slightly better than PointConv on most tasks and seem to generalize better to previously unseen queries. We show how PointConv can be augmented with an attention mechanism to bring its performance closer to interaction networks' performance without sacrificing intrepretability. However, it seems that in general if this interpretability is not needed, interaction networks are generally the most appropriate model architecture to apply to the spatio-temporal problems we evaluated.

Finally, we provide the codebase used to implement, define, perform, and evaluate every experiment presented in this work. The codebase is designed to be approachable and extensible, to allow and encourage interested parties to validate our results or modify and extend our experiments for further investigation.

References

Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. *Advances in neural information processing systems*, 29, 2016.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.

Max Horn, Michael Moor, Christian Bock, Bastian Rieck, and Karsten Borgwardt. Set functions for time series, 2019. URL `https://arxiv.org/abs/1909.12064`.

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

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

Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. *arXiv preprint arXiv:1707.01926*, 2017.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. *Advances in neural information processing systems*, 28, 2015.

Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pp. 464–472. IEEE, 2017.

Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

Wenxuan Wu, Zhongang Qi, and Li Fuxin. Pointconv: Deep convolutional networks on 3d point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9621–9630, 2019.

Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. *arXiv preprint arXiv:1709.04875*, 2017.

Qi Zhang, Jianlong Chang, Gaofeng Meng, Shiming Xiang, and Chunhong Pan. Spatio-temporal graph structure learning for traffic forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 1177–1185, 2020.