We present PyTorch Geometric Temporal, an open-source Python library for spatiotemporal machine learning. We designed PyTorch Geometric Temporal with a simple and consistent API inspired by the software architecture of existing widely used libraries, streamlined neural network layer definitions, temporal snapshot generators for batching, and integrated benchmark datasets. These features are illustrated with a tutorial-like case study. Experiments demonstrate the predictive performance of the models implemented in the library on real world problems such as epidemiological forecasting, ride-hail demand prediction and web-traffic management. Our sensitivity analysis of runtime shows that the framework can potentially operate on web-scale datasets with rich temporal features and spatial structure.

ABSTRACT

We present PyTorch Geometric Temporal, a deep learning framework combining state-of-the-art machine learning algorithms for neural spatiotemporal signal processing. The main goal of the library is to make temporal geometric deep learning available for researchers and machine learning practitioners in a unified easy-to-use framework. PyTorch Geometric Temporal was created with foundations on existing libraries in the PyTorch eco-system, streamlined neural network layer definitions, temporal snapshot generators for batching, and integrated benchmark datasets. These features are illustrated with a tutorial-like case study. Experiments demonstrate the predictive performance of the models implemented in the library on real world problems such as epidemiological forecasting, ride-hail demand prediction and web-traffic management. Our sensitivity analysis of runtime shows that the framework can potentially operate on web-scale datasets with rich temporal features and spatial structure.

1 INTRODUCTION

Deep learning on static graph structured data has seen an unprecedented success in various business and scientific application domains. Neural network layers which operate on graph data can serve as building blocks of document labeling, fraud detection, traffic forecasting and cheminformatics systems [7, 45–47, 63]. This emergence and the widespread adaptation of geometric deep learning was made possible by open-source machine learning libraries. The high quality, breadth, user-oriented nature and availability of specialized deep learning libraries [13, 15, 46, 67] were all contributing factors to the practical success and large-scale deployment of graph machine learning systems. At the same time the existing geometric deep learning frameworks operate on graphs which have a fixed topology and it is also assumed that the node features and labels are static. Besides limiting assumptions about the input data, these off-the-shelf libraries are not designed to operate on spatiotemporal data.

Present work. We propose PyTorch Geometric Temporal, an open-source Python library for spatiotemporal machine learning. We designed PyTorch Geometric Temporal with a simple and consistent API inspired by the software architecture of existing widely
used geometric deep learning libraries from the PyTorch ecosystem [15, 40]. Our framework was built by applying simple design principles consistently. The framework reuses existing neural network layers in a modular manner, models have a limited number of public methods and hyperparameters can be inspected. Spatiotemporal signal iterators ingest data memory efficiently in widely used scientific computing formats and return those in a PyTorch compatible format. The design principles in combination with the test coverage, documentation, practical tutorials, continuous integration, package indexing and frequent releases make the framework an end-user friendly spatiotemporal machine learning system.

The experimental evaluation of the framework entails node level regression tasks on datasets released exclusively with the framework. Specifically, we compare the predictive performance of spatiotemporal graph neural networks on epidemiological forecasting, demand planning, web traffic management and social media interaction prediction tasks. Synthetic experiments show that with the right batching strategy PyTorch Geometric Temporal is highly scalable and benefits from GPU accelerated computing.

**Our contributions.** The main contributions of our work can be summarized as:

- We publicly release PyTorch Geometric Temporal the first deep learning library for parametric spatiotemporal machine learning models.
- We provide data loaders and iterators with PyTorch Geometric Temporal which can handle spatiotemporal datasets.
- We release new spatiotemporal benchmark datasets from the renewable energy production, epidemiological reporting, goods delivery and web traffic forecasting domains.
- We evaluate the spatiotemporal forecasting capabilities of the neural and parametric machine learning models available in PyTorch Geometric Temporal on real world datasets.

The remainder of the paper has the following structure. In Section 2 we overview important preliminaries and the related work about temporal and geometric deep learning and the characteristics of related open-source machine learning software. The main design principles of PyTorch Geometric Temporal are discussed in Section 3 with a practical example. We demonstrate the forecasting capabilities of the framework in Section 4 where we also evaluate the scalability of the library on various commodity hardware. We conclude in Section 5 where we summarize the results. The source code of PyTorch Geometric Temporal is publicly available at https://github.com/benedekrozemberczki/pytorch_geometric_temporal, the Python package can be installed via the Python Package Index. Detailed documentation is accessible at https://pytorch-geometric-temporal.readthedocs.io/.

2 PRELIMINARIES AND RELATED WORK

In order to position our contribution and highlight its significance we introduce some important concepts about spatiotemporal data and discuss related literature about geometric deep learning and machine learning software.

2.1 Temporal Graph Sequences

Our framework considers specific input data types on which the spatiotemporal machine learning models operate. Input data types can differ in terms of the dynamics of the graph and that of the modelled vertex attributes. We take a discrete temporal snapshot view of this data representation problem [25, 26] and our work considers three spatiotemporal data types which can be described by the subplots of Figure 1 and the following formal definitions:

**Definition 2.1. Dynamic graph with temporal signal.** A dynamic graph with a temporal signal is the ordered set of graph and node feature matrix tuples \( D = \{(G_t, X_t), \ldots, (G_T, X_T)\} \) where the vertex sets satisfy \( V_t = V, \forall t \in \{1, \ldots, T\} \) and the node feature matrix that \( X_t \in \mathbb{R}^{V_t \times d}, \forall t \in \{1, \ldots, T\} \).

**Definition 2.2. Dynamic graph with static signal.** A dynamic graph with a static signal is the ordered set of graph and node feature matrix tuples \( D = \{(G_t, X), \ldots, (G_T, X)\} \) where vertex sets satisfy \( V_t = V, \forall t \in \{1, \ldots, T\} \) and the node feature matrix that \( X \in \mathbb{R}^{V \times d} \).

**Definition 2.3. Static graph with temporal signal.** A static graph with a temporal signal is the ordered set of graph and node feature matrix tuples \( D = \{(G, X_1), \ldots, (G, X_T)\} \) where the node feature matrix satisfies that \( X_t \in \mathbb{R}^{V \times d}, \forall t \in \{1, \ldots, T\} \).

Representing spatiotemporal data based on these theoretical concepts allows us the creation of memory efficient data structures which conceptualize these definitions in practice well.

![Dynamic graph with temporal signal.](image1.png)

(a) Dynamic graph with temporal signal.

![Dynamic graph with static signal.](image2.png)

(b) Dynamic graph with static signal.

![Static graph with temporal signal.](image3.png)

(c) Static graph with temporal signal.

Figure 1: The data iterators in PyTorch Geometric Temporal can provide temporal snapshots for all of the non static geometric deep learning scenarios.
2.2 Deep Learning with Time and Geometry

Our work provides deep learning models that operate on data which has both temporal and spatial aspects. These techniques are natural recombinations of existing neural network layers that operate on sequences and static graph-structured data.

2.2.1 Temporal Deep Learning. A large family of temporal deep learning models such as the LSTM [24] and GRU [12] generates in-memory representations of data points which are iteratively updated as it learns by new snapshots. Another family of deep learning models uses the attention mechanism [3, 35, 59] to learn representations of the data points which are adaptively recontextualized based on the temporal history. These types of models serve as templates for the temporal block of spatiotemporal deep learning models.

2.2.2 Static Graph Representation Learning. Learning representations of vertices, edges and whole graphs with graph neural networks in a supervised or unsupervised way can be described by the message passing formalism [17]. In this conceptual framework using the node and edge attributes in a graph as parametric function generates compressed representations (messages) which are propagated between the nodes based on a message-passing rule and aggregated to form new representations. Most of the existing graph neural network architectures such as GCN [30], GGCN[33], ChebyConv [14], and RGCN [50] fit perfectly into this general description of graph neural networks. Models are differentiated by assumptions about the input graph (e.g. node heterogeneity, multiplexity, presence of edge attributes), the message compression function used, the propagation scheme and message aggregation function applied to the received messages.

2.2.3 Spatiotemporal Deep Learning. A spatiotemporal deep learning model fuses the basic conceptual ideas of temporal deep learning techniques and graph representation learning. Operating on a temporal graph sequence these models perform message-passing at each time point with a graph neural network block and the new temporal information is incorporated by a temporal deep learning block. This design allows for sharing salient temporal and spatial autocorrelation information across the spatial units. The temporal and spatial layers which are fused together in a single parametric machine learning model are trained together jointly by exploiting the fact that the fused models are end-to-end differentiable. In Table 1 we summarized the spatiotemporal deep learning models implemented in the framework which we categorized based on the temporal and graph neural network layer blocks, the order of proximity considered and the heterogeneity of the edge set.

2.3 Graph Representation Learning Software

The current graph representation learning software ecosystem which allows academic research and industrial deployment extends open-source auto-differentiation libraries such as TensorFlow [1], PyTorch [41], MxNet [11] and JAX [16, 28]. Our work does the same as we build on the PyTorch Geometric ecosystem. We summarized the characteristics of these libraries in Table 2 which allows for comparing frameworks based on the backend, presence of supervised training functionalities, presence of temporal models and GPU support. Our proposed framework is the only one to date which allows the supervised training of temporal graph representation learning models with graphics card based acceleration.

Table 1: A comparison of spatiotemporal deep learning models in PyTorch Geometric Temporal based on the temporal and spatial block, order of proximity considered and the heterogeneity of the edges.

<table>
<thead>
<tr>
<th>Model</th>
<th>Temporal Layer</th>
<th>GNN Layer</th>
<th>Proximity Order</th>
<th>Multi Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCRNN [32]</td>
<td>GRU</td>
<td>DiffConv</td>
<td>Higher</td>
<td>False</td>
</tr>
<tr>
<td>GConvGRU [51]</td>
<td>GRU</td>
<td>Chebyshev</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>GConvLSTM [51]</td>
<td>LSTM</td>
<td>Chebyshev</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>GC-LSTM [10]</td>
<td>LSTM</td>
<td>Chebyshev</td>
<td>Lower</td>
<td>True</td>
</tr>
<tr>
<td>DyGrAE [54, 55]</td>
<td>LSTM</td>
<td>GGCN</td>
<td>Higher</td>
<td>False</td>
</tr>
<tr>
<td>LRGCN [31]</td>
<td>LSTM</td>
<td>RGCN</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>EGCN-H [39]</td>
<td>GRU</td>
<td>GCN</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>EGCN-O [39]</td>
<td>LSTM</td>
<td>GCN</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>T-GCN [65]</td>
<td>GRU</td>
<td>GCN</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>AST-GCN [68]</td>
<td>GRU</td>
<td>GCN</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>AGCRN [4]</td>
<td>GRU</td>
<td>Chebyshev</td>
<td>Higher</td>
<td>False</td>
</tr>
<tr>
<td>MPNN LSTM [38]</td>
<td>LSTM</td>
<td>GCN</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>STGCN [63]</td>
<td>Attention</td>
<td>Chebyshev</td>
<td>Higher</td>
<td>False</td>
</tr>
<tr>
<td>ASTGCN [22]</td>
<td>Attention</td>
<td>Chebyshev</td>
<td>Higher</td>
<td>False</td>
</tr>
<tr>
<td>MSTGCN [22]</td>
<td>Attention</td>
<td>Chebyshev</td>
<td>Higher</td>
<td>False</td>
</tr>
<tr>
<td>GMAN [66]</td>
<td>Attention</td>
<td>Custom</td>
<td>Lower</td>
<td>False</td>
</tr>
<tr>
<td>MTGNN [61]</td>
<td>Attention</td>
<td>Custom</td>
<td>Higher</td>
<td>False</td>
</tr>
<tr>
<td>AAGCN [52]</td>
<td>Attention</td>
<td>Custom</td>
<td>Higher</td>
<td>False</td>
</tr>
</tbody>
</table>

Table 2: A desiderata and automatic differentiation backend library based comparison of open-source geometric deep learning libraries.

<table>
<thead>
<tr>
<th>Library</th>
<th>Backend</th>
<th>Supervised</th>
<th>Temporal</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT Geometric [15]</td>
<td>PT</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>GeometricDPR [49]</td>
<td>PT</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CogDL [9]</td>
<td>PT</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Spektral [21]</td>
<td>TF</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>TF Geometric [27]</td>
<td>TF</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>StellarGraph [13]</td>
<td>TF</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DGL [67]</td>
<td>TF/PT/MX</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DFG [34]</td>
<td>PT</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Jraph [18]</td>
<td>JAX</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Graph-Learn [62]</td>
<td>Custom</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>GEM [19]</td>
<td>TF</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DynamicGEM [20]</td>
<td>TF</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>OpenNE [57]</td>
<td>Custom</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Karate Club [46]</td>
<td>Custom</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Our Work</td>
<td>PT</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

2.4 Spatiotemporal Data Analytics Software

The open-source ecosystem for spatiotemporal data processing consists of specialized database systems, basic analytical tools and
advanced machine learning libraries. We summarized the characteristics of the most popular libraries in Table 3 with respect to the year of release, purpose of the framework, source code language and GPU support.

First, it is evident that most spatiotemporal data processing tools are fairly new and there is much space for contributions in each subcategory. Second, the database systems are written in high-performance languages while the analytics and machine learning oriented tools have a pure Python/R design or a wrapper written in these languages. Finally, the use of GPU acceleration is not widespread which alludes to the fact that current spatiotemporal data processing tools might have a scalability issue. Our proposed framework PyTorch Geometric Temporal is the first fully open-source GPU accelerated spatiotemporal machine learning library.

### Table 3: A multi-aspect comparison of open-source spatiotemporal database systems, data analytics libraries and machine learning frameworks.

<table>
<thead>
<tr>
<th>Library</th>
<th>Year</th>
<th>Purpose</th>
<th>Language</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeoWave [60]</td>
<td>2016</td>
<td>Database</td>
<td>Java</td>
<td>✗</td>
</tr>
<tr>
<td>StaCSpec [23]</td>
<td>2017</td>
<td>Database</td>
<td>Javascript</td>
<td>✗</td>
</tr>
<tr>
<td>MobilityDB [69]</td>
<td>2019</td>
<td>Database</td>
<td>C</td>
<td>✗</td>
</tr>
<tr>
<td>PyStac [44]</td>
<td>2020</td>
<td>Database</td>
<td>Python</td>
<td>✗</td>
</tr>
<tr>
<td>StaRs [42]</td>
<td>2017</td>
<td>Analytics</td>
<td>R</td>
<td>✗</td>
</tr>
<tr>
<td>CuSpatial [56]</td>
<td>2019</td>
<td>Analytics</td>
<td>Python</td>
<td></td>
</tr>
<tr>
<td>PySal [43]</td>
<td>2017</td>
<td>Machine Learning</td>
<td>Python</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Our work</strong></td>
<td>2021</td>
<td>Machine Learning</td>
<td>Python</td>
<td>✔</td>
</tr>
</tbody>
</table>

3. **THE FRAMEWORK DESIGN**

Our primary goal is to give a general theoretical overview of the framework, discuss the framework design choices, give a detailed practical example and highlight our strategy for the long term viability and maintenance of the project.

3.1 **Neural Network Layer Design**

The spatiotemporal neural network layers are implemented as classes in the framework. Each of the classes has a similar architecture driven by a few simple design principles.

3.1.1 **Non-proliferation of classes.** The framework reuses the existing high level neural network layer classes as building blocks from the PyTorch and PyTorch Geometric ecosystems. The goal of the library is not to replace the existing frameworks. This design strategy makes sure that the number of auxiliary classes in the framework is kept low and that the framework interfaces well with the rest of the ecosystem.

3.1.2 **Hyperparameter inspection and type hinting.** The neural network layers do not have default hyperparameter settings as some of these have to be set in a dataset dependent manner. In order to help with this, the layer hyperparameters are stored as public class attributes and they are available for inspection. Moreover, the constructors of the neural network layers use type hinting which helps the end-users to set the hyperparameters.

3.1.3 **Limited number of public methods.** The spatiotemporal neural network layers in our framework have a limited number of public methods for simplicity. For example, the auxiliary layer initialization methods and other internal model mechanics are implemented as private methods. All of the layers provide a forward method and those which explicitly use the message-passing scheme in PyTorch Geometric provide a public message method.

3.1.4 **Auxiliary layers.** The auxiliary neural network layers which are not part of the PyTorch Geometric ecosystem such as diffusion convolutional graph neural networks [32] are implemented as standalone neural network layers in the framework. These layers are available for the design of novel neural network architectures as individual components.

3.2 **Data Structures**

The design of PyTorch Geometric Temporal required the introduction of custom data structures which can efficiently store the datasets and provide temporally ordered snapshots for batching.

3.2.1 **Spatiotemporal Signal Iterators.** Based on the categorization of spatiotemporal signals discussed in Section 2 we implemented three types of Spatiotemporal Signal Iterators. These iterators store spatiotemporal datasets in memory efficiently without redundancy. For example a Static Graph Temporal Signal iterator will store the edge indices and weights for each time period in order to save memory. By iterating over a Spatiotemporal Signal Iterator at each step a graph snapshot is returned which describes the graph of interest at a given point in time. Graph snapshots are returned in temporal order by the iterators. The Spatiotemporal Signal Iterators can be indexed directly to access a specific graph snapshot – a design choice which allows the use of advanced temporal batching.

3.2.2 **Graph Snapshots.** The time period specific snapshots which consist of labels, features, edge indices and weights are stored as NumPy arrays [58] in memory, but returned as a PyTorch Geometric Data object instance [15] by the Spatiotemporal Signal Iterators when these are iterated on. This design choice hedges against the proliferation of classes and exploits the existing and widely used compact data structures from the PyTorch ecosystem [40].

3.2.3 **Train-Test Splitting.** As part of the library we provide a temporal train-test splitting function which creates train and test snapshot iterators from a Spatiotemporal Signal Iterator given a test dataset ratio. This parameter of the splitting function decides the fraction of data that is separated from the end of the spatiotemporal graph snapshot sequence for testing. The returned iterators have the same type as the input iterator. Importantly, this splitting does not influence the applicability of widely used semi-supervised model training strategies such as node masking.

3.2.4 **Integrated Benchmark Dataset Loaders.** We provided easy-to-use practical data loader classes for widely used existing [38] and the newly released benchmark datasets. These loaders return Spatiotemporal Signal Iterators which can be used for training existing and custom designed spatiotemporal neural network architectures to solve supervised machine learning problems.
3.3 Design in Practice Case Study: Cumulative Model Training on CPU

In the following we overview a simple end-to-end machine learning pipeline designed with PyTorch Geometric Temporal. These code snippets solve a practical epidemiological forecasting problem – predicting the weekly number of chickenpox cases in Hungary [47]. The pipeline consists of data preparation, model definition, training and evaluation phases.

```python
from torch_geometric_temporal import ChickenpoxDatasetLoader
from torch_geometric_temporal import temporal_signal_split
loader = ChickenpoxDatasetLoader()
dataset = loader.get_dataset()
train, test = temporal_signal_split(dataset, train_ratio=0.9)
```

Listings 1: Loading a default benchmark dataset and creating a temporal split with PyTorch Geometric Temporal

3.3.1 Dataset Loading and Splitting. In Listings 1 as a first step we import the Hungarian chickenpox cases benchmark dataset loader and the temporal train test splitter function (lines 1-2). We define the dataset loader (line 4) and use the `get_dataset()` class method to return a temporal signal iterator (line 5). Finally, we create a train-test split of the spatiotemporal dataset by using the splitting function and retain 10% of the temporal snapshots for model performance evaluation (lines 7-8).

```python
import torch.nn.functional as F
from torch_geometric_temporal.nn.recurrent import DCRNN
class RecurrentGCN(torch.nn.Module):
    def __init__(self, node_features, filters):
        super(RecurrentGCN, self).__init__()
        self.recurrent = DCRNN(node_features, filters, 1)
        self.linear = torch.nn.Linear(filters, 1)
    def forward(self, x, edge_index, edge_weight):
        h = self.recurrent(x, edge_index, edge_weight)
        h = self.linear(h)
        return h
```

Listings 2: Defining a recurrent graph convolutional neural network using PyTorch Geometric Temporal consisting of a diffusion convolutional spatiotemporal layer followed by rectified linear unit activations, dropout and a feedforward neural network layer.

3.3.2 Recurrent Graph Convolutional Model Definition. We define a recurrent graph convolutional neural network model in Listings 2. We import the base and functional programming PyTorch libraries and one of the neural network layers from PyTorch Geometric Temporal (lines 1-3). The model requires a node feature count and convolutional filter parameter in the constructor (line 6). The model consists of a one-hop Diffusion Convolutional Recurrent Neural Network layer [32] and a fully connected layer with a single neuron (lines 8-9).

In the forward pass method of the neural network the model uses the vertex features, edges and the optional edge weights (line 11). The initial recurrent graph convolution based aggregation (line 12) is followed by a rectified linear unit activation function [37] and dropout [53] for regularization (lines 13-14). Using the fully-connected layer the model outputs a single score for each spatial unit (lines 15-16).

```python
model = RecurrentGCN(node_features=8, filters=32)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
for epoch in range(200):
    cost = 0
    for time, snapshot in enumerate(train):
        y_hat = model(snapshot.x, snapshot.edge_index, snapshot.edge_attr)
        cost = cost + torch.mean((y_hat-snapshot.y)**2)
        cost = cost / (time+1)
        optimizer.step()
    model.eval()
    cost = 0
    for time, snapshot in enumerate(test):
        y_hat = model(snapshot.x, snapshot.edge_index, snapshot.edge_attr)
        cost = cost + torch.mean((y_hat-snapshot.y)**2)
        cost = cost / (time+1)
        if cost > 0.01
            print("MSE: {:.4f}".format(cost))
```

Listings 3: Evaluating the recurrent graph convolutional neural network on the test portion of the spatiotemporal dataset using the time unit averaged mean squared error.

3.3.3 Model Training. Using the dataset split and the model definition we can turn our attention to training a regressor. In Listings 3 we create a model instance (line 1), transfer the model parameters (line 3) to the Adam optimizer [29] which uses a learning rate of 0.01 and set the model to be trainable (line 5). In each epoch we set the accumulated cost to be zero (line 8) iterate over the temporal snapshots in the training data (line 9), make forward passes with the model on each temporal snapshot and accumulate the spatial unit specific mean squared errors (lines 10-13). We normalize the cost, backpropagate and update the model parameters (lines 14-17).

```python
model.eval()
for time, snapshot in enumerate(train):
    y_hat = model(snapshot.x, snapshot.edge_index, snapshot.edge_attr)
    cost = cost + torch.mean((y_hat-snapshot.y)**2)
    cost = cost / (time+1)
    if cost > 0.01
        print("MSE: {:.4f}".format(cost))
```

3.4 Design in Practice Case Study: Incremental Model Training with GPU Acceleration

Exploiting the power of GPU based acceleration of computations happens at the training and evaluation steps of the PyTorch Geometric Temporal pipelines. In this case study we assume that the Hungarian Chickenpox cases dataset is already loaded in memory, the temporal split happened and a model class was defined by the code snippets in Listings 1 and 2. Moreover, we assume that the machine used for training the neural network can access a single CUDA compatible GPU device [48].

```python
model = RecurrentGCN(node_features=8, filters=32)
device = torch.device('cuda')
model = model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
model.train()
for epoch in range(200):
    for snapshot in train:
        snapshot = snapshot.to(device)
        y_hat = model(snapshot.x,
                      snapshot.edge_attr)
        cost = torch.mean((y_hat-snapshot.y)**2)
        cost.backward()
        optimizer.step()
        optimizer.zero_grad()
```

Listings 5: Creating a recurrent graph convolutional neural network instance and training it by incremental weight updates on a GPU.

3.4.1 Model Training. In Listings 5 we demonstrate accelerated training with incremental weight updates. The model of interest and the device used for training are defined while the model is transferred to the GPU (lines 1-3). The optimizer registers the model parameters and the model parameters are set to be trainable (lines 5-6). We iterate over the temporal snapshot iterator 200 times and the iterator returns a temporal snapshot in each step. Importantly the snapshots which are PyTorch Geometric Data objects are transferred to the GPU (lines 8-10). The use of PyTorch Geometric Data objects as temporal snapshots allows the transfer of the time period specific edges, node features and target vector with a single command. Using the input data a forward pass is made, loss is accumulated and weight updates happen using the optimizer in each time period (lines 11-17). Compared to the cumulative backpropagation based training approach discussed in Subsection 3.3 this backpropagation strategy is slower as weight updates happen at each time step, not just at the end of training epochs.

3.4.2 Model Evaluation. During model scoring the GPU can be utilized again. The snippet in Listings 6 demonstrates that the only modification needed for accelerated evaluation is the transfer of snapshots to the GPU. In each time period we move the temporal snapshot to the device to do the forward pass (line 4). We do the forward pass with the model and the snapshot on the GPU and accumulate the loss (lines 5-8). The loss value is averaged out and detached from the GPU for printing (lines 9-11).

```python
model.eval()
cost = 0
for time, snapshot in enumerate(test):
    snapshot = snapshot.to(device)
    y_hat = model(snapshot.x,
                  snapshot.edge_index,
                  snapshot.edge_attr)
    cost = cost + torch.mean((y_hat-snapshot.y)**2)
    cost = cost / (time+1)
cost = cost.item()
pred = torch.argmax(y_hat, dim=1)
pred = (pred == snapshot.y).float()
print("Accuracy: {:.2f}%".format(100*torch.mean(pred)))
```

Listings 6: Evaluating the recurrent graph convolutional neural network with GPU based acceleration.

3.5 Maintaining PyTorch Geometric Temporal

The viability of the project is made possible by the open-source code, version control, public releases, automatically generated documentation, continuous integration, and near 100% test coverage.

3.5.1 Open-Source Code-Base and Public Releases. The source code of PyTorch Geometric Temporal is publicly available on GitHub under the MIT license. Using an open version control system allowed us to have a large group collaborate on the project and have external contributors who also submitted feature requests. The public releases of the library are also made available on the Python Package Index, which means that the framework can be installed via the `pip` command using the terminal.

3.5.2 Documentation. The source code of PyTorch Geometric Temporal and Sphinx [8] are used to generate a publicly available documentation of the library at https://pytorch-geometric-temporal.readthedocs.io/. This documentation is automatically created every time when the code-base changes in the public repository. The documentation covers the constructors and public methods of neural network layers, temporal signal iterators, public dataset loaders and splitters. It also includes a list of relevant research papers, an in-depth installation guide, a detailed getting-started tutorial and a list of integrated benchmark datasets.

3.5.3 Continuous Integration. We provide continuous integration for PyTorch Geometric Temporal with GitHub Actions which are available for free on GitHub without limitations on the number of builds. When the code is updated on any branch of the repository the build process is triggered and the library is deployed on Linux, Windows and macOS virtual machines.
3.5.4 Unit Tests and Code Coverage. The temporal graph neural network layers, custom data structures and benchmark dataset loaders are all covered by unit tests. These unit tests can be executed locally using the source code. Unit tests are also triggered by the continuous integration provided by GitHub Actions. When the master branch of the open-source GitHub repository is updated, the build is successful, and all of the unit tests pass a coverage report is generated by CodeCov.

4 EXPERIMENTAL EVALUATION

The proposed framework is evaluated on node level regression tasks using novel datasets which we release with the paper. We also evaluate the effect of various batching techniques on the predictive performance and runtime.

4.1 Datasets

We release new spatiotemporal benchmark datasets with PyTorch Geometric Temporal which can be used to test models on node level regression tasks. The descriptive statistics and properties of these newly introduced benchmark datasets are summarized in Table 4.

| Dataset                      | Signal  | Graph    | Frequency | 𝑇   | |𝑉   |
|------------------------------|---------|----------|-----------|-----|-----|
| Chickenpox Hungary           | Temporal| Static   | Weekly    | 522 | 20  |
| Windmill Large               | Temporal| Static   | Hourly    | 17,472| 319 |
| Windmill Medium              | Temporal| Static   | Hourly    | 17,472| 26  |
| Windmill Small               | Temporal| Static   | Hourly    | 17,472| 11  |
| Pedal Me Deliveries          | Temporal| Static   | Weekly    | 36  | 15  |
| Wikipedia Math               | Temporal| Static   | Dynamic   | 731 | 1,068|
| Twitter Tennis RG            | Static  | Dynamic  | Hourly    | 120 | 1000|
| Twitter Tennis UO            | Static  | Dynamic  | Hourly    | 112 | 1000|
| Covid19 England              | Temporal| Dynamic  | Hourly    | 61  | 129 |
| Montevideo Buses             | Temporal| Static   | Hourly    | 744 | 675 |
| MTM-1 Hand Motions           | Temporal| Static   | 1/24 Seconds| 14,469| 21  |

These newly released datasets are the following:

- **Chickenpox Hungary.** A spatiotemporal dataset about the officially reported cases of chickenpox in Hungary. The nodes are counties and edges describe direct neighbourhood relationships. The dataset covers the weeks between 2005 and 2015 without missingness.
- **Windmill Output Datasets.** An hourly windfarm energy output dataset covering 2 years from a European country. Edge weights are calculated from the proximity of the windmills – high weights imply that two windmill stations are in close vicinity. The size of the dataset relates to the grouping of windfarms considered; the smaller datasets are more localized to a single region.
- **Pedal Me Deliveries.** A dataset about the number of weekly bicycle package deliveries by Pedal Me in London during 2020 and 2021. Nodes in the graph represent geographical units and edges are proximity based mutual adjacency relationships.
- **Twitter Tennis RG and UO.** Twitter mention graphs of major tennis tournaments from 2017. Each snapshot contains the graph of popular player or sport news accounts and mentions between them [5, 6]. Node labels encode the number of mentions received and vertex features are structural properties.
- **Covid19 England.** A dataset about mass mobility between regions in England and the number of confirmed COVID-19 cases from March to May 2020 [38]. Each day contains a different mobility graph and node features corresponding to the number of cases in the previous days. Mobility stems from Facebook Data For Good 1 and cases from gov.uk 2.
- **Montevideo Buses.** A dataset about the hourly passenger inflow at bus stop level for ten bus lines from the city of Montevideo. Nodes are bus stops and edges represent connections between the stops; the dataset covers a whole month of traffic patterns.
- **MTM-1 Hand Motions.** A temporal dataset of Methods-Time Measurement-1 [36] motions, signalled as consecutive graph frames of 21 3D hand key points that were acquired via MediaPipe Hands [64] from original RGB-Video material. Node features encode the normalized xyz-coordinates of each finger joint and the vertices are connected according to the human hand structure.

4.2 Predictive Performance

The forecasting experiments focus on the evaluation of the recurrent graph neural networks implemented in our framework. We compare the predictive performance under two specific backpropagation regimes which can be used to train these recurrent models:

- **Incremental:** After each temporal snapshot the loss is backpropagated and model weights are updated. This would need as many weight updates as the number of temporal snapshots.
- **Cumulative:** When the loss from every temporal snapshot is aggregated it is backpropagated and weights are updated with the optimizer. This requires one weight update per epoch.

4.2.1 Experimental settings. Using 90% of the temporal snapshots for training, we evaluated the forecasting performance on the last 10% by calculating the average mean squared error from 10 experimental runs. We used models with a recurrent graph convolutional layer which had 32 convolutional filters. The spatiotemporal layer was followed by the rectified linear unit [37] activation function and during training time we used a dropout of 0.5 for regularization [53] after the spatiotemporal layer. The hidden representations were fed to a fully connected feedforward layer which outputted the predicted scores for each spatial unit. The recurrent models

1https://dataforgood.fb.com/
2https://coronavirus.data.gov.uk/
were trained for 100 epochs with the Adam optimizer [29] which used a learning rate of $10^{-2}$ to minimize the mean squared error.

4.2.2 Experimental findings. Results are presented in Table 5 where we also report standard deviations around the test set mean squared error and bold numbers denote the best performing model under each training regime on a dataset. Our experimental findings demonstrate multiple important empirical regularities which have important practical implications. Namely these are the following:

1. Most recurrent graph neural networks have a similar predictive performance on these regression tasks. In simple terms there is not a single model which acts as silver bullet. This also postulates that the model with the lowest training time is likely to be as good as the slowest one.

2. Results on the Wikipedia Math dataset imply that a cumulative backpropagation strategy can have a detrimental effect on the predictive performance of a recurrent graph neural network. When computation resources are not a bottleneck an incremental strategy can be significantly better.

4.3 Runtime Performance

The evaluation of the PyTorch Geometric Temporal runtime performance focuses on manipulating the input size and measuring the time needed to complete a training epoch. We investigate the runtime under the incremental and cumulative backpropagation strategies.

4.3.1 Experimental settings. The runtime evaluation used the GConvGru model [51] with the hyperparameter settings described in Subsection 4.2. We measured the time needed for doing a single epoch over a sequence of 100 synthetic graphs. Reference Watts-Strogatz graphs in the snapshots of the dynamic graph with temporal signal iterator had binary labels, $2^{10}$ nodes, $2^{5}$ edges per node and $2^{3}$ node features. Runtimes were measured on the following hardware:

- CPU: The machine used for benchmarking had 8 Intel 1.00 GHz i5-1035G1 processors.
- GPU: We utilized a machine with a single Tesla V-100 graphics card for the experiments.

![Figure 2: The average time needed for doing an epoch on a dynamic graph – temporal signal iterator of Watts Strogatz graphs with a recurrent graph convolutional model.](image)

Table 5: The predictive performance of spatiotemporal neural networks evaluated by average mean squared error. We report average performances calculated from 10 experimental repetitions with standard deviations around the average mean squared error calculated on 10% forecasting horizons. We use the incremental and cumulative backpropagation strategies.

<table>
<thead>
<tr>
<th></th>
<th>Chickenpox Hungary</th>
<th>Twitter Tennis RG</th>
<th>PedalMe London</th>
<th>Wikipedia Math</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Incremental</td>
<td>Cumulative</td>
<td>Incremental</td>
<td>Cumulative</td>
</tr>
<tr>
<td>DCRNN [52]</td>
<td>1.124 ± 0.015</td>
<td>1.123 ± 0.014</td>
<td>1.463 ± 0.019</td>
<td>1.450 ± 0.024</td>
</tr>
<tr>
<td>GConvGRU [51]</td>
<td>1.128 ± 0.011</td>
<td>1.132 ± 0.023</td>
<td>1.622 ± 0.032</td>
<td>1.944 ± 0.013</td>
</tr>
<tr>
<td>GConvLSTM [51]</td>
<td>1.121 ± 0.014</td>
<td>1.129 ± 0.027</td>
<td>1.442 ± 0.028</td>
<td>1.533 ± 0.020</td>
</tr>
<tr>
<td>GC-LSTM [10]</td>
<td>1.115 ± 0.014</td>
<td>1.116 ± 0.023</td>
<td>1.455 ± 0.023</td>
<td>1.468 ± 0.025</td>
</tr>
<tr>
<td>DyGeAE [54, 55]</td>
<td>1.120 ± 0.021</td>
<td>1.118 ± 0.015</td>
<td>2.031 ± 0.006</td>
<td>2.007 ± 0.004</td>
</tr>
<tr>
<td>EGCN-H [39]</td>
<td>1.113 ± 0.016</td>
<td>1.104 ± 0.024</td>
<td>1.467 ± 0.026</td>
<td>1.436 ± 0.017</td>
</tr>
<tr>
<td>EGCN-O [39]</td>
<td>1.124 ± 0.009</td>
<td>1.118 ± 0.020</td>
<td>1.491 ± 0.024</td>
<td>1.430 ± 0.023</td>
</tr>
<tr>
<td>AST-GCN [68]</td>
<td>1.114 ± 0.008</td>
<td>1.119 ± 0.018</td>
<td>1.469 ± 0.027</td>
<td>1.475 ± 0.029</td>
</tr>
<tr>
<td>T-GCN [65]</td>
<td>1.117 ± 0.011</td>
<td>1.111 ± 0.022</td>
<td>1.479 ± 0.012</td>
<td>1.481 ± 0.029</td>
</tr>
<tr>
<td>MPN LSTM [38]</td>
<td>1.116 ± 0.023</td>
<td>1.129 ± 0.021</td>
<td>1.485 ± 0.028</td>
<td>1.458 ± 0.013</td>
</tr>
<tr>
<td>AGCRN [4]</td>
<td>1.120 ± 0.010</td>
<td>1.116 ± 0.017</td>
<td>2.039 ± 0.022</td>
<td>2.010 ± 0.009</td>
</tr>
</tbody>
</table>

- CPU: The machine used for benchmarking had 8 Intel 1.00 GHz i5-1035G1 processors.
- GPU: We utilized a machine with a single Tesla V-100 graphics card for the experiments.
5 CONCLUSIONS AND FUTURE DIRECTIONS

In this paper we discussed PyTorch Geometric Temporal the first deep learning library designed for neural spatiotemporal signal processing. We reviewed the existing geometric deep learning and machine learning techniques implemented in the framework. We gave an overview of the general machine learning framework design principles, the newly introduced input and output data structures, long-term project viability and discussed a case study with source-code which utilized the library. Our empirical evaluation focused on (a) the predictive performance of the models available in the library on real world datasets which we released with the framework; (b) the scalability of the methods under various input sizes and structures.

Our work could be extended and it also opens up opportunities for novel geometric deep learning and applied machine learning research. A possible direction to extend our work would be the consideration of temporal models which operate on curved spaces such as hyperbolic and spherical spaces. We are particularly interested in how the spatiotemporal deep learning techniques in the framework can be deployed and used for solving high-impact practical machine learning tasks.

REFERENCES


Aynaz Taheri and Tanya Berger-Wolf. 2019. Predictive Temporal Embedding of
Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory
PyTorch: An Imperative Style, High-Performance Deep Learning Library. In

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory
PyTorch: An Imperative Style, High-Performance Deep Learning Library. Advances in

https://github.com/r-spatial/stars.

Sergio J Rey and Luc Anselin. 2010. PySAL: A Python Library of Spatial Analytical

Emanuele Rob. 2020. PySTAC: Python library for working with any SpatioTempo-

Benedek Rozemberczki, Peter Englert, Anmol Kapoor, Martin Blais, and Bryan

Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. 2020. Karate Club: An
API Oriented Open-source Python Framework for Unsupervised Learning on
Knowledge Management. 3125–3132.

Benedek Rozemberczki, Paul Scherer, Oliver Kiss, Rik Sarkar, and Tamas Ferenci.
2021. Chickenpox Cases in Hungary: a BenchmarkDataset for Spatiotemporal
Signal Processing with Graph Neural Networks. arXiv:2102.08100 [cs.LG]

Jason Sanders and Edward Kandrot. 2010. CUDA by Example: An Introduction to
General-Purpose GPU Programming. Addison-Wesley Professional.

with GeoZDR. In IML Workshop on Graph Representation Learning and Beyond.

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan
Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolu-

Youngjo Seo, Michael Defferrard, Pierre Vandergheynst, and Xavier Bresson.
2018. Structured Sequence Modeling with Graph Convolutional Recurrent Net-
works. In International Conference on Neural Information Processing. Springer,
362–373.

Lei Shi, Yifan Zhang, Jian Cheng, and Hanqing Lu. 2019. Two-Stream Adaptive
Graph Convolutional Networks for skeleton-Based Action Recognition. In Pro-
cedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.
12026–12035.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan
Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from

Aynaz Taheri and Tanya Berger-Wolf. 2019. Predictive Temporal Embedding of
on Advances in Social Networks Analysis and Mining. 57–64.

Aynaz Taheri, Kevin Gimpel, and Tanya Berger-Wolf. 2019. Learning to Represen-
t the Evolution of Dynamic Graphs with Recurrent Models. In Companion

Paul Taylor, Christopher Harris, Thompson Comer, and Mark Harris. 2019. CUDA-
cuspatial.

Cunchao Tu, Yuan Yao, Zhengyuan Zhang, Ganqu Cui, Hao Wang, Changxin Tian,
Embedding. https://github.com/huans/OpenNE.

Stephen Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy
Computing in science &

ing Distributed Key-Value Stores for Multidimensional Data. In International

Zonghan Wu, Shirui Pan, Guodong Long, Jiang Jiang, Xiaojuan Chang, and Chengqi
Graph Neural Networks. In Proceedings of the 26th ACM SIGKDD International
Conference on Knowledge Discovery & Data Mining. 753–763.

Hongxia Yang. 2019. AlGraph: A Comprehensive Graph Neural Network Plat-
form. In Proceedings of the 25th ACM SIGKDD International Conference on Knowl-
edge Discovery & Data Mining. 3165–3166.

Bing Yu, Haoteng Yin, and Zhanxing Zhu. 2018. Spatio-Temporal Graph Con-
volutional Networks: a Deep Learning Framework for Traffic Forecasting. In Pro-
cedings of the 27th International Joint Conference on Artificial Intelligence. 3634–3640.

Fan Zhang, Valentin Bazarevsky, Andrey Vakunov, Andrei Tkachenko, George
Sung, Chao-Ling Chang, and Matthias Grundmann. 2020. MediaPipe Hands: