Dynamic Node Property Prediction on Temporal Reddit Graph

Team: Popov Nikita and Izotova Anastasiia

Dataset

tgbn-reddit. This is a users and subreddits interaction network. Both users and subreddits are nodes and each edge indicates that a user posted on a subreddit at a given time. The dataset spans from 2005 to 2019. The task is to learn the interaction frequency towards the subreddits of a user over the next week.

	ts	user	subreddit	num_words	score
0	1199145604	52820223	reddit.com	4	4
1	1199145650	51035265	politics	19	4
2	1199145698	56552862	reddit.com	39	2
3	1199145728	5369809	politics	44	1
4	1199145733	2775893	reddit.com	127	3

	ts	user	subreddit	weight
0	1199232004	52820223	reddit.com	0.834177
1	1199232004	52820223	science	0.021519
2	1199232004	52820223	programming	0.144304
3	1199232004	51035265	politics	0.855629
4	1199232004	51035265	reddit.com	0.144371

Dynamic Node Property Prediction

The goal of dynamic node property prediction is to predict the property of a node at any given timestamp t, i.e., to learn a function f : Vt → Y, where Vt is the set of nodes at time t and Y is some output space (e.g. {−1, +1}, R, Rp , etc). In our case the task is to learn the interaction frequency towards the subreddits of a user for the next week.

Node affinity prediction:

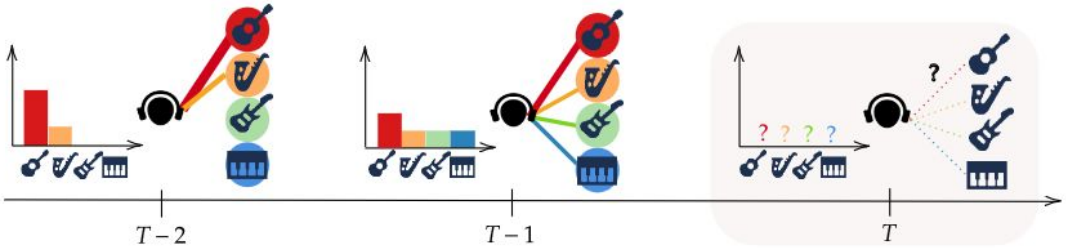


Figure 3: The *node affinity prediction* task aims to predict how the preference of a user towards items change over time. In the `tgbn-genre` example, the task is to predict the frequency at which the user would listen to each genre over the next week given their listening history until today.

Persistent forecast

Simple baseline for time series forecasting

The main idea is to return the last seen label of the node (or zero vector if it doesn't exist) for the current time t

```python
class PersistentForecaster:
    def __init__(self, num_class):
        self.dict = {}
        self.num_class = num_class

    def update_dict(self, node_id, label):
        self.dict[node_id] = label

    def query_dict(self, node_id):

        #node_id: the node to query
        if node_id in self.dict:
            return self.dict[node_id]
        else:
            return np.zeros(self.num_class)
```

Moving average

Model for considering the average of the node labels observed in the previous K steps

The basic idea is to return the average of the last seen node labels (or zero vector if it doesn't exist) for the current time t

```
class MovingAverage:
    def __init__(self, num_class, K=7):
        self.dict = {}
        self.num_class = num_class
        self.K = K

    def update_dict(self, node_id, label):
        if node_id in self.dict:
            total = self.dict[node_id] * (self.K - 1) + label
            self.dict[node_id] = total / self.K
        else:
            self.dict[node_id] = label

    def query_dict(self, node_id):

        #node_id: the node to query
        if node_id in self.dict:
            return self.dict[node_id]
        else:
            return np.zeros(self.num_class)
```

TGN

Memory: The memory of a node is updated after an event (e.g. interaction with another node or node-wise change), and its purpose is to represent the node's history in a compressed format.

Message: For each event involving node i, a message is computed to update i's memory.

$$\mathbf{m}_i(t) = \mathrm{msg_s}\left(\mathbf{s}_i(t^-), \mathbf{s}_j(t^-), \Delta t, \mathbf{e}_{ij}(t)\right), \qquad \mathbf{m}_j(t) = \mathrm{msg_d}\left(\mathbf{s}_j(t^-), \mathbf{s}_i(t^-), \Delta t, \mathbf{e}_{ij}(t)\right)$$

$\mathbf{s}_i(t^-)$ is the memory of node i just before time t.

msg - in general are learnable functions. We choose the message function as identity (id), which is simply the concatenation of the inputs, for the sake of simplicity.

TGN

Message Aggregator - a mechanism to aggregate messages to certain node from batch.

$$\bar{\mathbf{m}}_i(t) = \text{agg}\left(\mathbf{m}_i(t_1), \ldots, \mathbf{m}_i(t_b)\right)$$

For the sake of simplicity we considered most recent message (keep only most recent message for a given node)

Memory Updater. The memory of a node is updated upon each event involving the node itself

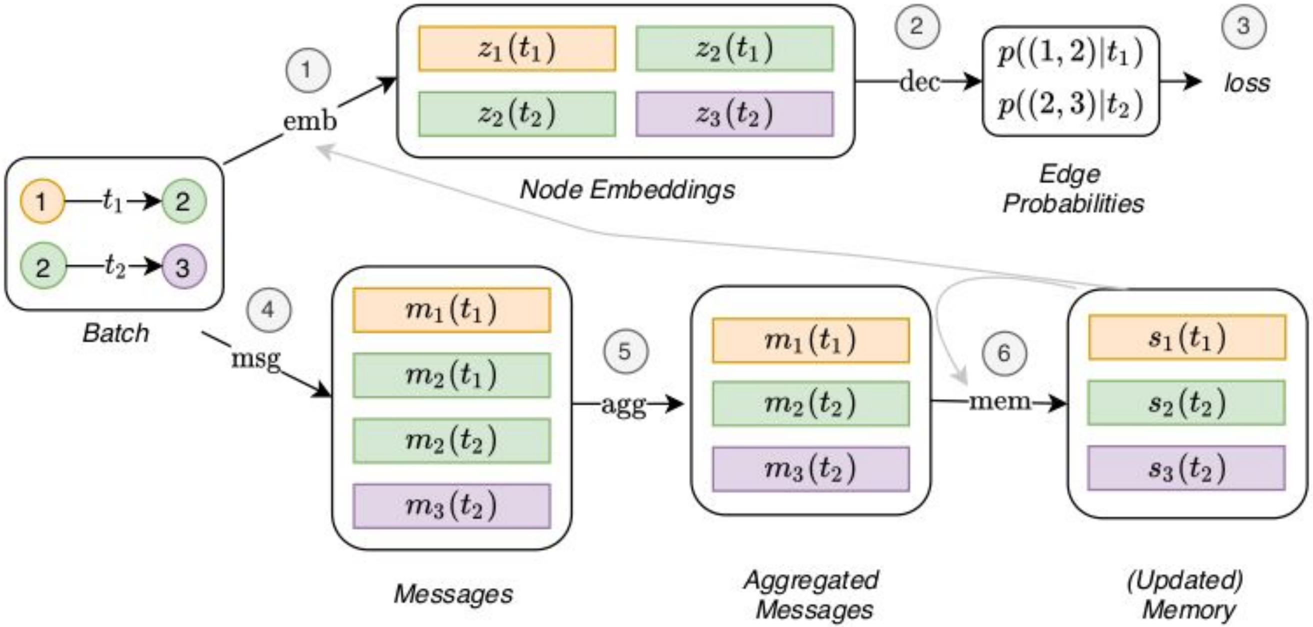
$$\mathbf{s}_i(t) = \text{mem}\left(\bar{\mathbf{m}}_i(t), \mathbf{s}_i(t^-)\right)$$

TGN

Embedding. The embedding module is used to generate the temporal embedding zi (t) of node i at any time t. Here we use graph attention embeddings.

$$
\begin{aligned}
\mathbf{h}_i^{(l)}(t) &= \mathrm{MLP}^{(l)}(\mathbf{h}_i^{(l-1)}(t) \,\|\, \tilde{\mathbf{h}}_i^{(l)}(t)), \\
\tilde{\mathbf{h}}_i^{(l)}(t) &= \mathrm{MultiHeadAttention}^{(l)}(\mathbf{q}^{(l)}(t), \mathbf{K}^{(l)}(t), \mathbf{V}^{(l)}(t)), \\
\mathbf{q}^{(l)}(t) &= \mathbf{h}_i^{(l-1)}(t) \,\|\, \phi(0), \\
\mathbf{K}^{(l)}(t) &= \mathbf{V}^{(l)}(t) = \mathbf{C}^{(l)}(t), \\
\mathbf{C}^{(l)}(t) &= [\mathbf{h}_1^{(l-1)}(t) \,\|\, \mathbf{e}_{i1}(t_1) \,\|\, \phi(t - t_1), \ \ldots, \ \mathbf{h}_N^{(l-1)}(t) \,\|\, \mathbf{e}_{iN}(t_N) \,\|\, \phi(t - t_N)]
\end{aligned}
$$

TGN



1
emb
$z_1(t_1)$
$z_2(t_1)$
$z_2(t_2)$
$z_3(t_2)$
Node Embeddings
2
dec
$p((1,2)|t_1)$
$p((2,3)|t_2)$
Edge Probabilities
3
loss
1
t_1
2
2
t_2
3
Batch
4
msg
$m_1(t_1)$
$m_2(t_1)$
$m_2(t_2)$
$m_3(t_2)$
Messages
5
agg
$m_1(t_1)$
$m_2(t_2)$
$m_3(t_2)$
Aggregated Messages
6
mem
$s_1(t_1)$
$s_2(t_2)$
$s_3(t_2)$
(Updated) Memory

PyG functionalities for TGN

models.TGNMemory

class **TGNMemory** (**num_nodes**: int, **raw_msg_dim**: int, **memory_dim**: int, **time_dim**: int, **message_module**: Callable, **aggregator_module**: Callable) [source]

Bases: `Module`

The Temporal Graph Network (TGN) memory model from the "Temporal Graph Networks for Deep Learning on Dynamic Graphs" paper.

> **Note**
>
> For an example of using TGN, see examples/tgn.py.

PARAMETERS:

- **num_nodes** (*int*) – The number of nodes to save memories for.
- **raw_msg_dim** (*int*) – The raw message dimensionality.
- **memory_dim** (*int*) – The hidden memory dimensionality.
- **time_dim** (*int*) – The time encoding dimensionality.
- **message_module** (*torch.nn.Module*) – The message function which combines source and destination node memory embeddings, the raw message and the time encoding.
- **aggregator_module** (*torch.nn.Module*) – The message aggregator function which aggregates messages to the same destination into a single representation.

Also torch_geometric.nn .models.tgn contains classes for aggregate messages and get last neigbours of node

PyG functionalities

conv.TransformerConv

class **TransformerConv** (**in_channels**: Union[int, Tuple[int, int]], **out_channels**: int, **heads**: int = 1, **concat**: bool = True, **beta**: bool = False, **dropout**: float = 0.0, **edge_dim**: Optional[int] = None, **bias**: bool = True, **root_weight**: bool = True, ****kwargs**) [source]

Bases: `MessagePassing`

The graph transformer operator from the "Masked Label Prediction: Unified Message Passing Model for Semi-Supervised Classification" paper.

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} \mathbf{W}_2 \mathbf{x}_j,$$

where the attention coefficients $\alpha_{i,j}$ are computed via multi-head dot product attention:

$$\alpha_{i,j} = \mathrm{softmax}\left(\frac{(\mathbf{W}_3\mathbf{x}_i)^\top (\mathbf{W}_4\mathbf{x}_j)}{\sqrt{d}} \right)$$

```
class TemporalDataLoader(torch.utils.data.DataLoader):
    r"""A data loader which merges succesive events of a
    :class:`torch_geometric.data.TemporalData` to a mini-batch.

    Args:
        data (TemporalData): The :obj:`~torch_geometric.data.TemporalData`
            from which to load the data.
        batch_size (int, optional): How many samples per batch to load.
            (default: :obj:`1`)
        neg_sampling_ratio (float, optional): The ratio of sampled negative
            destination nodes to the number of postive destination nodes.
            (default: :obj:`0.0`)
        **kwargs (optional): Additional arguments of
            :class:`torch.utils.data.DataLoader`.
```

TGN code

```python
class GraphAttentionEmbedding(torch.nn.Module):
    """
    Reference:
    - https://github.com/pyg-team/pytorch_geometric/blob/master/examples/tgn.py
    """

    def __init__(self, in_channels, out_channels, msg_dim, time_enc):
        super().__init__()
        self.time_enc = time_enc
        edge_dim = msg_dim + time_enc.out_channels
        self.conv = TransformerConv(
            in_channels, out_channels // 2, heads=2, dropout=0.1, edge_dim=edge_dim
        )

    def forward(self, x, last_update, edge_index, t, msg):
        rel_t = last_update[edge_index[0]] - t
        rel_t_enc = self.time_enc(rel_t.to(x.dtype))
        edge_attr = torch.cat([rel_t_enc, msg], dim=-1)
        return self.conv(x, edge_index, edge_attr)
```

```python
#decoder
class NodePredictor(torch.nn.Module):
    def __init__(self, in_dim, out_dim):
        super().__init__()
        self.lin_node = Linear(in_dim, in_dim)
        self.out = Linear(in_dim, out_dim)

    def forward(self, node_embed):
        h = self.lin_node(node_embed)
        h = h.relu()
        h = self.out(h)
        # h = F.log_softmax(h, dim=-1)
        return h
```

Results

Normalized Discounted Cumulative Gain (nDCG) metric

	train	val	test
Persistent forecast	0.37	0.38	0.37
Moving average	0.58	0.57	0.56
TGN	0.43	0.34	0.31

References

1. Temporal Graph Networks for Deep Learning on Dynamic Graphs
   https://arxiv.org/abs/2006.10637
2. Temporal Graph Benchmark for Machine Learning on Temporal Graphs
   https://arxiv.org/abs/2307.01026
3. https://tgb.complexdatalab.com/