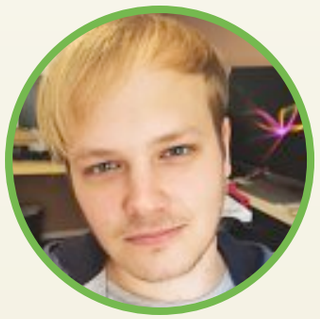
dortmund university

Stanford University

SFB 876 Providing Information by Resource-Constrained Data Analysis

GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings

Matthias Fey



Jan E. Lenssen

Frank Weichert

Jure Leskovec

matthias.fey@udo.edu

/rusty1s/pyg_autoscale

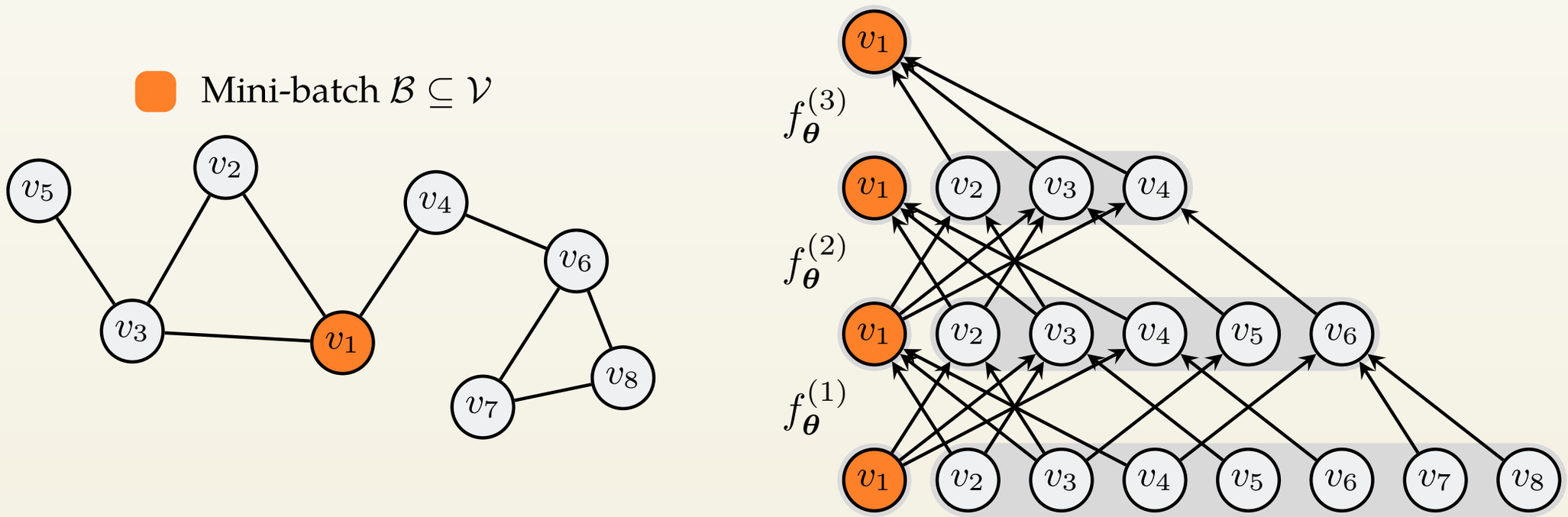
Scalable Graph Neural Networks

Applying **Graph Neural Networks**

$$\mathbf{h}_v^{(\ell+1)} = f_\theta^{(\ell+1)}\left(\mathbf{h}_v^{(\ell)}, \left\{\!\!\left\{ \mathbf{h}_w^{(\ell)} : w \in \mathcal{N}(v) \right\}\!\!\right\}\right)$$

to **large-scale graphs** is challenging due to the **"neighbor explosion"** problem

- *exponentially increasing dependency of nodes over layers*



Scalable Graph Neural Networks

The most common approaches for scaling up GNNs work by **sampling edges**

Hamilton et al., 2017; Chen et al., 2018; Zou et al., 2019; Huang et al., 2018; Chiang et al., 2019; Zeng et al. 2019;...

However, just by the act of sampling edges, a GNN **fails to learn** anything about **structural graph properties**

- *this leads to reduced model expressivity!*

How can we learn structural graph properties while still being scalable? 🤔

Historical Embeddings

We can utilize **historical embeddings** to **approximate** the **missing out-of-mini-batch information** for each layer

$$
\begin{aligned}
\mathbf{h}_v^{(\ell+1)} &= f_\theta^{(\ell+1)}\left(\mathbf{h}_v^{(\ell)}, \left\{\!\!\left\{ \mathbf{h}_w^{(\ell)} : w \in \mathcal{N}(v) \right\}\!\!\right\}\right) \\
&= f_\theta^{(\ell+1)}\left(\mathbf{h}_v^{(\ell)}, \left\{\!\!\left\{ \mathbf{h}_w^{(\ell)} : w \in \mathcal{N}(v) \cap \mathcal{B} \right\}\!\!\right\} \cup \left\{\!\!\left\{ \mathbf{h}_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B} \right\}\!\!\right\}\right) \\
&\approx f_\theta^{(\ell+1)}\left(\mathbf{h}_v^{(\ell)}, \left\{\!\!\left\{ \mathbf{h}_w^{(\ell)} : w \in \mathcal{N}(v) \cap \mathcal{B} \right\}\!\!\right\} \cup \underbrace{\left\{\!\!\left\{ \bar{\mathbf{h}}_w^{(\ell)} : w \in \mathcal{N}(v) \setminus \mathcal{B} \right\}\!\!\right\}}_{\text{Historical Embeddings}}\right)
\end{aligned}
$$

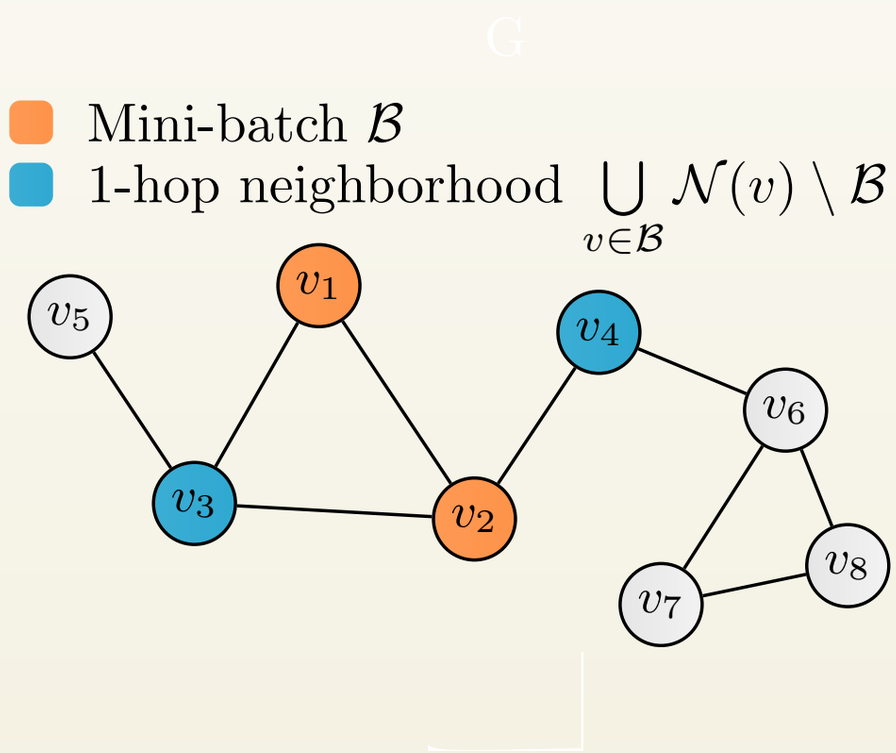
Historical embeddings represent node embeddings ***acquired in previous training iterations***

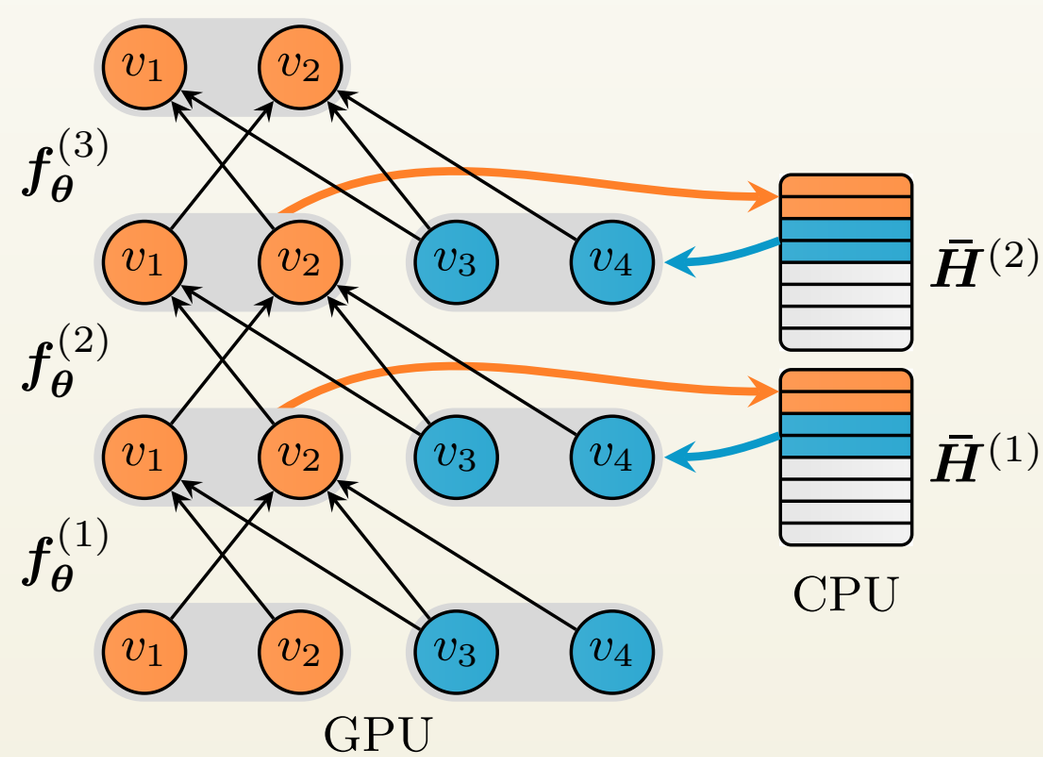
A **generalization of the work** of Chen et al., 2018

Historical Embeddings

We can utilize **historical embeddings** to **approximate** the **missing out-of-mini-batch information** for each layer

- *We pull the most recent histories from out-of-mini-batch nodes*
- *We push newly estimated embeddings to histories*





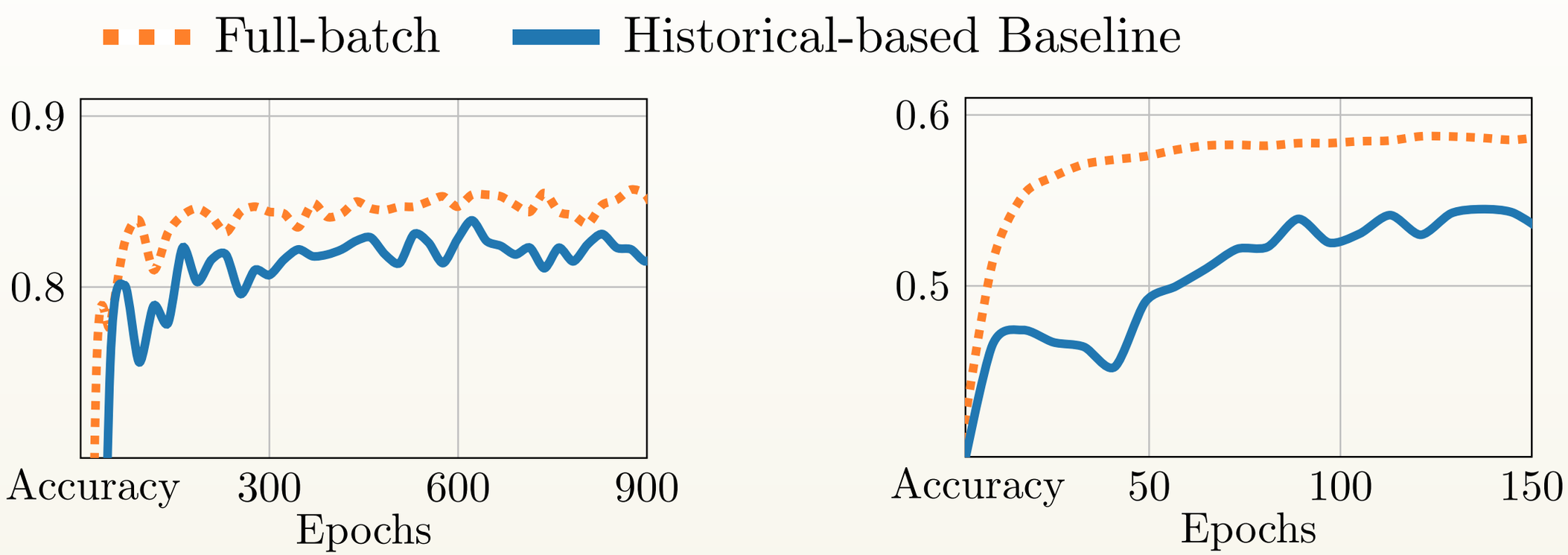
Theoretical Analysis

A historical-based GNN makes use of **all available neighborhood information:**

- *Its approximation error is solely bounded by ...*
  1. *the **staleness of histories***
  2. *the **Lipschitz continuity** of the GNN's message functions*
  3. *the **number of layers***

- *It can provably be **as expressive as the WL-test** in distinguishing non-isomorphic subgraphs* 🤗

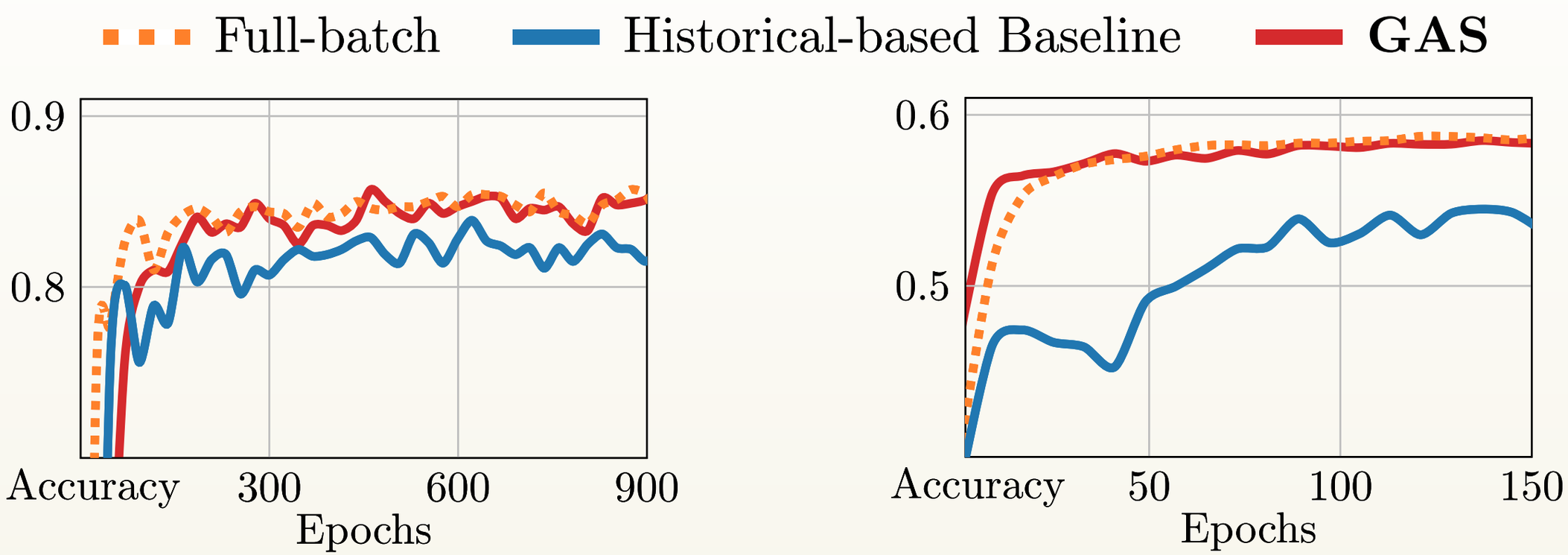
Tightening Error Bounds



(a) 64-layer GCNII

(b) 4-layer GIN

Tightening Error Bounds



(a) 64-layer `GCNII` (b) 4-layer `GIN`

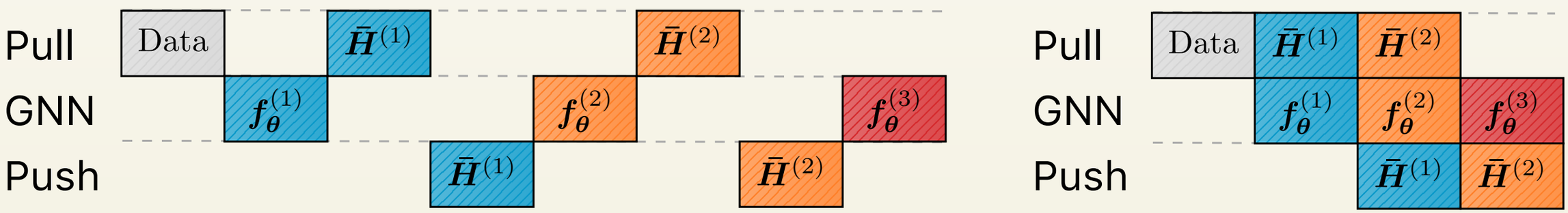
In practice, we **need to tighten the error bound** for **deep** or **expressive** GNN:

1. Reducing the amount of history accesses via **clustering**
2. Enforcing Lipschitz continuity via **regularization**

Efficient History Accesses

- **frequent data transfers** to and from the GPU can cause **major I/O bottlenecks** 😭
- We use **non-blocking device transfers** to counteract 🤗

1. Immediately start transferring history chunks **asynchronously** at the start of forward execution
2. **Synchronize** individual CUDA stream before GPU access



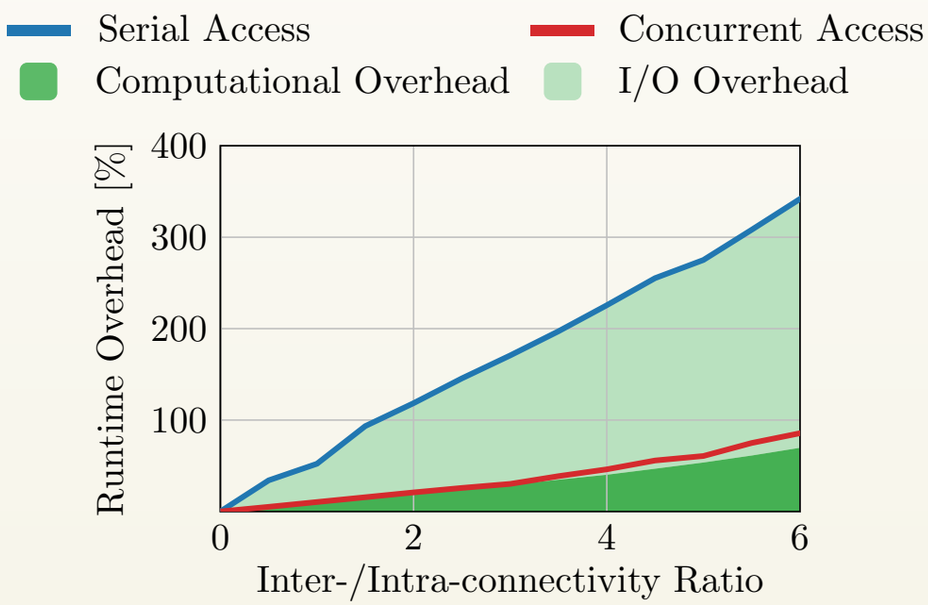
(a) Serial execution **(b)** Concurrent execution

Experimental Evaluation

Our GAS framework is ...

- *fast and memory efficient*
- *has no runtime overhead induced by history accesses*
- *can be applied to large-scale graphs with any GNN backbone*

Dataset	Runtime (s)		Memory (MB)	
	GTTF	**GAS**	GTTF	**GAS**
CORA	0.077	**0.006**	18.01	**2.13**
PUBMED	0.071	**0.006**	28.79	**2.19**
PPI	0.976	**0.007**	134.86	**12.37**
FLICKR	1.178	**0.007**	325.97	**16.32**



	# nodes	230K	57K	89K	717K	169K	2.4M
	# edges	11.6M	794K	450K	7.9M	1.2M	61.9M
	Method	REDDIT	PPI	FLICKR	YELP	ogbn-arxiv	ogbn-products
GAS	GCN	95.45	98.92	54.00	62.94	71.68	76.66
GAS	GCNII	96.77	**99.50**	56.20	65.14	**73.00**	77.24
GAS	PNA	**97.17**	99.44	**56.67**	64.40	72.50	**79.91**

Conclusion

- ✓ **constant GPU memory** consumption *w.r.t.* input node size
- ✓ able to **reason about graph structures** at scale
- ✓ (*nearly*) **no runtime overhead** induced by history accesses
- ✓ can be applied with **any GNN backbone**
- ✓ **fully open-sourced** at /`rusty1s/pyg_autoscale` on top of PyG

```
~ class GNN(ScalableGNN):
    def __init__(self, ...):
~     super().__init__(num_nodes, hidden_channels, num_layers)
      self.conv1 = GCNConv(...)
      self.conv2 = GCNConv(...)

~   def forward(self, x, edge_index, *args):
        x = self.conv1(x, edge_index).relu()
+       x = self.push_and_pull(self.histories[0], x, *args)
        x = self.conv2(x, edge_index)
        return x
```