CatGNN: Category Theory-based GNN Library

Karolis Špukas

1 Introduction

Neural algorithmic reasoning is a way to execute algorithmic computations using neural networks in a high-dimensional latent space without the need to manually convert between natural inputs and algorithmic inputs [VB21]. Deciding on the architecture of such networks is not a trivial task. It has been shown that architectures aligning better to the algorithms they are emulating give better performance, the so-called algorithmic alignment principle [Xu+19]. Inspired by the algorithmic alignment of graph neural networks (GNNs) and dynamic programming algorithms (DPs) [Xu+19], a recent work by [DV22] proposed a brand new framework to generalise both GNNs and DP algorithms using category theory and abstract algebra. This framework is thought to be capable of expressing both GNN computations and DP algorithms through a new set of primitives: pullbacks, kernels, pushforwards and aggregators. Reasoning about these primitives has already allowed the authors to verify a known result of GNN and DP alignment - the choice of the aggregator. An explicit implementation of the new framework would provide further evidence to the correctness of results and could give rise to a new way of developing GNNs.

The main goal of this mini-project is to investigate possible category theory-based GNN library design directions. Three different implementations of a generic Message-Passing GNN (MPNN) layer are proposed, each with varying level of abstraction and performance. Using this template, several well-known GNN layers, such as graph convolution networks (GCN) [KW16] and graph attention networks (GAT) [Vel+17], are implemented and evaluated on standard benchmark datasets. Performance results (training, validation and test accuracy at each epoch) indicate correct implementation, though training times are slightly slower than in leading "traditional" GNN libraries like PyTorch Geometric [FL19]. It is expected that the mini-project will be useful when deciding on the best implementation of the new framework and act as a further evidence of its correctness. Source code can be found on GitHub.[1]

2 Background

Consider a graph $G = (V, E)$ where V are vertices and E are edges. Suppose node features are stored in a node feature matrix $\mathbf{X} \in \mathbb{R}^{|V| \times k}$. Then a generic message-passing GNN (MPNN) layer over G looks as follows:

$$\mathbf{h}_u = \phi\left(\mathbf{x}_u, \bigoplus_{v \in \mathcal{N}_u} \psi(\mathbf{x}_u, \mathbf{x}_v)\right) \tag{1}$$

where $\mathbf{x}_i \in \mathbf{X}$ is the feature of node i, $\mathcal{N}_i$ is the set of one-hop neighbours of node i, $\psi : \mathbb{R}^k \times \mathbb{R}^k \to \mathbb{R}^k$ and $\phi : \mathbb{R}^k \to \mathbb{R}^k$ are some functions (say MLP) and $\bigoplus$ is a permutation-invariant aggregation function. [DV22] has recently proposed a new framework where any such GNN can alternatively be represented using a generic diagram of an integral transform. We can express the same graph G as a span:

$$V \xleftarrow{s} E \xrightarrow{t} V \tag{2}$$

[1]GitHub repository: https://github.com/KaroliShp/CatGNN

where V, E are sets of vertices and edges, respecitvely, and s, t are functions that give sender and receiver nodes for each edge $e \in E$, respectively. Then, given initial features $f : V \to R$ where R is a commutative monoid, we obtain updated features $f' : V \to R$ as follows:

$$[V, R] \xrightarrow{s^*} [E, R] \xrightarrow{k} [E, R] \xrightarrow{t_*} [V, \mathbb{N}[R]] \xrightarrow{\bigoplus} [V, R] \tag{3}$$

where s^* is the pullback operator, k is the kernel transformation, t_* is the pushforward operator and $\bigoplus$ is the aggregation operator. When receiver-dependent features are desired, the above diagram can be expanded further by factorizing k:

$$[V, R] \xrightarrow{s^*} [E, R] \xrightarrow{(\mathrm{id},*)} [E, R] \times [E, R] \xrightarrow{k'} [E, R] \xrightarrow{t_*} [V, \mathbb{N}[R]] \xrightarrow{\bigoplus} [V, R] \tag{4}$$

Using these four primitives (and sticking an appropriate update function ϕ on top), we ought to be able to express any MPNN as defined in equation 1. Specifically:

- Pullback s^* corresponds to "pulling" sender node features into edge features
- Message function ψ corresponds to the kernel transformation k using pulled features to create edge messages
- Pushforward t_* corresponds to collecting edge messages from all senders into a single "bag" of values
- Permutation-invariant aggregation function $\bigoplus$ correspond to the aggregator primitive $\bigoplus$

For a visual representation of integral transform, refer to Figure 1 in [DV22].

3 API overview

In this section, we give a high-level overview of the API. Many development decisions were inspired by PyG, mainly to ease adaptability for the end-user. This also significantly sped up API prototyping and benchmarking without needing to "reinvent the wheel" at the early stages of development process. For more technical details (Python version, testing, code formatting) refer to `README.md` on GitHub.

3.1 Graph representation

Graphs are treated similarly to PyG. Each graph $G = (V, E, \mathbf{X})$ consists of a set of vertices V (tensor `torch.int64`), a set of edges E (tensor `torch.int64`) and a feature matrix $\mathbf{X}$. Each vertex in V must have a unique integer index in range $[0, |V| - 1]$. Edges E are represented in COOrdinate (COO) format, i.e. E has a shape `(2,-1)`. Each row index of the node feature matrix $\mathbf{X}$ must correspond to the vertex index, e.x. features in the first row of $\mathbf{X}$ must correspond to the node with index 0. Such graph representation is in close agreement with the description given in the paper, as any sets V, E and corresponding features can be easily transformed into this representation. It also allows seamless integration with existing PyG utilities like minibatching.

For the sake of simplicity, we make several assumptions. Firstly, R is real-valued vectors, which is typical for GNNs. In accordance with the paper, we also only deal with node features, ignoring any edge/graph feature updates. While this limits the GNNs we can currently recreate within the framework, this level of complexity is not deemed necessary at this stage of development. Lastly, while E could be of type `SparseTensor` like in PyG, we limit ourselves to simply `Tensor`. Since both use COO format, memory consumption remains the same and we did not have to deal with sparse tensor operations.

3.2 Minibatching, datasets, benchmarks

Due to limited time, important miscellaneous utilities, such as minibatching and datasets, are imported and adapted from PyG. It already provides most datasets that have been typically used in the literature to benchmark GNNs. Semi-supervised learning benchmark code, i.e. Cora, CiteSeer and PubMed, was written by us, while code for graph classification and OGB was adapted from existing sources. Relevant references are present in code comments. If deemed necessary, these utilities could be reimplemented in CatGNN at a later stage of development process.

3.3 Integral transform

Integral transform is at the heart of this API. Its four arrows, the new set of primitives, are the building blocks of a generic GNN. The user is provided with a generic base class `BaseMPNNLayer`. Similarly to PyG's `MessagePassing` interface, it extends `torch.nn.Module`. As a result, we are able to directly use PyTorch backpropagation utilities and easily integrate with any PyTorch code to build complex models right off the bat.

To implement any GNN, the user simply has to extend the base class and provide the implementation of the four primitives $s^*, k, t_*, \oplus$, i.e. "insert lambdas with the right type". `BaseMPNNLayer` also provides the user with additional methods $s : E \to V, t : E \to V, t^{-1} : V \to P(E)$ and $f : V \to R$, which are required to implement the four arrows. To sum up, user experience looks a lot like PyG, but with a different set of primitives. Integral transform code can be found in `catgnn/integral_transform`.

4 Integral transform implementation

One can think of various ways to implement integral transform for GNNs. The main question is how to do that efficiently (*performance*), while remaining close to the paper's description (*level of abstraction*) and offering good user experience.

4.1 Naive implementation (BaseMPNNLayer_1)

We start with the most basic or "naive" implementation of a generic GNN layer, which is closest to the original paper but has performance issues. We call it `BaseMPNNLayer_1`. The idea is simple: since we use Python, we can treat functions as objects and build the integral transform compositionally. For example, consider pullback $s^* : [V, R] \to [E, R]$, specifically its "canonical" version $s^*(e) = f(s(e))$. It takes in a function f in $[V, R]$ and gives back a function in $[E, R]$. User implementation of this primitive may look as follows:

```python
def define_pullback(self, f):
    # Input f: V -> R
    # Return pullback: E -> R
    def pullback(e):
        # Input e \in E
        # Return r \in R
        return f(self.s(e)) # User implementation
    return pullback
```

This definition simply makes sense: `define_pullback()` takes in a function `f` in $[V, R]$ and returns a function `pullback()` in $[E, R]$. A function in $[E, R]$ can be defined by defining what happens for each $e \in E$. The user implementation also exactly follows paper's description where `f(self.s(e))` is the same as $f(s(e))$. We extend this example to the other primitives. For instance, $k : [E, R] \to [E, R]$ takes in the output of s^* (function in $[E, R]$) and gives back another function in $[E, R]$. The following code snippet illustrates how a simple kernel that leaves pulled features unchanged may be implemented by the user:

```python
def define_kernel(self, pullback):
    # Input pullback: E -> R
    # Return kernel: E -> R
    def kernel(e):
        # Input e \in E
        # Return r \in R
        return pullback(e) # User implementation
    return kernel
```

We can combine these primitive functions into a single pipeline and execute integral transform. To do that, the user simply has to call `BaseMPNNLayer.transform()` on some graph, similarly to PyG's `MessagePassing.propagate()` method. Behind the scenes, its implementation is straightforward:

```python
class BaseMPNNLayer_1(torch.nn.Module):
    ...
    def transform(self, V, E, X):
        ...
        # This is the gist of integral transform
        pullback = self.define_pullback(self.f) # Input V -> R, output E -> R
        kernel = self.define_kernel(pullback) # Input E -> R, output E -> R
        pushforward = self.define_pushforward(kernel) # Input E -> R, output V -> N[R]
        aggregator = self.define_aggregator(pushforward) # Input V -> N[R], output V -> R

        # Apply the constructed function (aggregator) to each node in the graph
        for v in V:
            # Call aggregator(v) and obtain aggregated_features for all v \in V

        # Before returning, call update function \phi on top if needed
        return self.update(X, aggregated_features)
    ...
```

Sparing the details, the user has to fill in the following template to implement any MPNN (we only consider sender features in this example, but implementation of factorized kernel arrow also exists):

```python
class CustomMPNNLayer(BaseMPNNLayer_1):
    def __init__(self):
        super().__init__()
        # Extra user implementation

    def forward(self, V, E, X):
        # Extra user implementation
        # User must call self.transform() to execute integral transform
        return self.transform(V, E, X)

    def define_pullback(self, f):
        def pullback(e):
            # User implementation
        return pullback

    def define_kernel(self, pullback):
        def kernel(e):
            # User implementation
        return kernel

    def define_pushforward(self, kernel):
        def pushforward(v):
            # User implementation
        return pushforward

    def define_aggregator(self, pushforward):
        def aggregator(v):
            # User implementation
        return aggregator

    def update(self, X, output):
        # User implementation
```

As mentioned, this approach closely follows the paper and may be considered "human-friendly", since we are used to defining functions by defining what they do for each input $e \in E$. However, its performance is abysmal as expected due to looping through tensors and a number of function calls. It is also not trivial to implement the utility functions s, t, t^{-1}, f.

4.2 Improved implementation (BaseMPNNLayer_2)

It is not hard to see that the naive implementation faces significant performance issues (exact figures later). The only way to solve them is to remove loops by vectorization and avoid function calls by using sets of vertices/edges as arguments. That is, functions should take tensors of edges/vertices as inputs rather than individual edges/vertices, and both user and API implementations should perform tensor operations on them. For example, consider the following improved "canonical" pullback s^* where the input is some set of edges E rather than an individual edge e:

```
def define_pullback(self, f):
    # Input f: V -> R
    # Return pullback: E -> R
    def pullback(E):
        # Input E (set of edges)
        # Output R (set of values for each edge)
        return f(self.s(E)) # User implementation
    return pullback
```

The benefits of this approach are instantly obvious when we consider how clean `transform()` looks:

```
class BaseMPNNLayer_2(torch.nn.Module):
    ...
    def transform(self, V, E, X):
        ...
        pullback = self.define_pullback(self.f) # Input V -> R, output E -> R
        kernel = self.define_kernel(pullback) # Input E -> R, output E -> R
        pushforward = self.define_pushforward(kernel) # Input E -> R, output V -> N[R]
        aggregator = self.define_aggregator(pushforward) # Input V -> N[R], output V -> R

        # Apply the constructed function (aggregator) to each node in the graph
        aggregated_features = aggregator(V)
        return self.update(X, aggregated_features)
    ...
```

This construction also allows the user to use any subset of V and E, which should come in handy when dealing with GNNs that choose their own spans.

While kernel transformation definition remains pretty much the same, the hardest primitive to efficiently define this way is the pushforward t_* and subsequent aggregator $\oplus$. The major problem with t_* is efficiently finding and storing preimages of all nodes of V in a single tensor, i.e. implementing t^{-1}. Each node can have a differently sized preimage, which makes it infeasible to simply store it in a single tensor. Having experimented with various approaches like Python `lists`, we ended up with creating a new `bag_index` tensor alongside the tensor E for performance reasons. `bag_index` holds the index of the bag for which a specific edge in E belongs to. For example, given a set of edges $0 \to 1$ and $0 \to 2$, we know that the first edge $0 \to 1$ belongs to the bag of values for the node 1, while the second edge $0 \to 2$ belongs to the bag of values for the node 2. Therefore, its `bag_index` is $[1, 2]$. This construction also allows the user to efficiently implement aggregator using `torch_scatter` or more advanced `pytorch_scatter`[2], similarly to PyG.

For a more elaborate explanation of how `BaseMPNNLayer_2` and each of the utility functions $s, t, t^{-1}, f, *$ are implemented, see Appendix A. For an explanation of how a GCN layer can be implemented using this template from user's perspective, see Appendix B. `BaseMPNNLayer_2` is the main integral transform implementation in CatGNN and is used for all experiments.

4.3 "Forwards" implementation

In this section, we provide an alternative way to build integral transform. This implementation offers identical performance (accuracy and training time) as `BaseMPNNLayer_2`, however it provides the user

[2] https://github.com/rusty1s/pytorch_scatter

with a completely different way to implement the same primitives. We call it "forwards" implementation, while the BaseMPNNLayer_2 implementation in section 4.2 is called the standard or "backwards" implementation due to its recursive nature.

Forwards implementation is given for the sake of being an alternative that the user may prefer. However, we personally prefer the standard (backwards) BaseMPNNLayer_2 implementation as it is more flexible and closer to the paper's description. Forwards implementation is more in style of PyG. It is also generally not as well thought out, especially when it comes to GNNs that choose their own spans. Moreover, some features like factorisation of the kernel arrow have not been implemented. Depending on the feedback, this alternative may be completely removed from the library.

The main idea of forwards implementation is to build the integral transform more intuitively, similar to how it is described in Figure 1 in [DV22]. Start by iterating over all the edges and pull back their sender features. Then apply kernel transform to obtain edge messages. Lastly, iterate over all nodes to collect their bags of values using pushforward, and aggregate. User implementation of canonical pullback could look as follows:

```python
def pullback(self, E, f, s):
    return f(s(E)) # User implementation
```

Primitive methods look much simpler as f, s are passed as regular parameters. However, this comes at a cost of not closely following paper's description. Finally, integral transform would be executed by calling `transform_forwards()` method on a given graph. Its implementation is given below for comparison with `transform()` in section 4.2:

```python
class BaseMPNNLayer_2(torch.nn.Module):
    ...
    def transform_forwards(self, V, E, X):
        ...
        # Pull node features into edge features for all edges
        pulledback_features = self.pullback(E, self.f, self.s)

        # Construct edge messages for all edges
        edge_messages = self.kernel(E, pulledback_features)

        # For each node, collect the edge messages into bags
        edge_messages, bag_indices = self.pushforward(V, edge_messages, self.t, self.t_1)

        # Aggregate for selected V
        aggregated_output = self.aggregator(V, edge_messages, bag_indices)

        # Update and return
        return self.update(X, aggregated_output)
    ...
```

Notice that the input to the next primitive is no longer a function, but rather the output of the previous primitive.

5 Empirical evaluation

In this section, we evaluate the performance of CatGNN and compare it to PyG. We are interested in test accuracy, which indicates correct implementation, and training time. Empirical evaluation consists of two parts. Firstly, we benchmark the two APIs on standard semi-supervised node classification and supervised graph classification tasks (Cora, MUTAG etc.). This is a good sanity check and an indication of expected performance differences. Network architectures, hyperparameters and experiment setups are identical to PyG paper [FL19] and are identical for both CatGNN and PyG. After that, we benchmark some of the smaller datasets from Open Graph Benchmark (OGB) [Hu+20] using implementations from the OGB leaderboard. These datasets are more realistic and larger, giving a better picture of CatGNN's capabilities. All experiments are performed on Tesla K80 (12GB GPU) in Google Colab, relevant notebooks are uploaded to GitHub. However, notebook output is not included, since experiments were carried out on multiple accounts over several days due to resource restrictions on Google Colab.

Note: even when running PyG implementation with the code provided by PyG, we could not always obtain the exact figures reported in [FL19]. We also observed that both average and standard deviation of final test accuracy depend heavily on the run for some of the graph classification tasks.

5.1 Semi-supervised node classification

Following GNN literature and [FL19], we use standard semi-supervised node classification datasets: Cora, CiteSeer and PubMed [Sen+08]. We use two types of layers: ones that use sender features only, namely GCN [KW16] and SGC [Wu+19], and ones that use both sender and receiver features, namely GAT [Vel+17]. Experiment setups are identical to [FL19], though custom benchmarking code was written by us. Results are presented in Table 1. Average and standard deviation are calculated over 100 runs for the public train/val/test splits.

Model	**Cora**		**CiteSeer**		**PubMed**	
	Accuracy/%	Time/s	Accuracy/%	Time/s	Accuracy/%	Time/s
GCN_1	$\sim$80.6	7.000	$\sim$70.8	10.000	-	80.000
GCN_2	80.6 ± 0.7	0.013	70.8 ± 0.8	0.017	79.2 ± 0.4	0.025
GCN_2 (factored)	80.6 ± 0.7	0.015	70.8 ± 0.8	0.020	79.1 ± 0.4	0.030
GCN_2 (forwards)	80.6 ± 0.7	0.013	70.9 ± 0.9	0.016	79.2 ± 0.4	0.024
PyG GCN	$\mathbf{80.7 \pm 0.7}$	**0.005**	$\mathbf{70.8 \pm 0.7}$	**0.005**	$\mathbf{79.1 \pm 0.5}$	**0.006**
SGC_2	80.3 ± 0.2	0.087	71.6 ± 0.2	0.218	74.8 ± 0.3	0.257
PyG SGC	$\mathbf{80.3 \pm 0.2}$	**0.018**	$\mathbf{71.6 \pm 0.2}$	**0.042**	$\mathbf{74.8 \pm 0.3}$	**0.049**
GAT_2	79.4 ± 2.4	0.025	67.7 ± 1.6	0.041	77.1 ± 0.6	0.059
PyG GAT	$\mathbf{78.4 \pm 3.2}$	**0.008**	$\mathbf{66.5 \pm 3.4}$	**0.009**	$\mathbf{76.2 \pm 1.9}$	**0.009**

Table 1: Semi-supervised node classification. Accuracy columns refer to test accuracy. Time columns refer to the average time taken to execute a single training epoch. Subscripts next to CatGNN models refer to the BaseMPNNLayer implementation, e.x. GCN_2 refers to the standard (backwards) implementation of BaseMPNNLayer_2. PyG refers to PyG implementation. GCN and SGC layers are uncached in both PyG and CatGNN, as caching was not implemented in CatGNN due to time constraints. GAT only uses 1 attention head because user implementation of more heads has not been optimized.

Note that we only use the improved standard (backwards) implementation (`BaseMPNNLayer_2`) for all experiments. The "naive" implementation (`BaseMPNNLayer_1`) is simply too slow; we only give its performance for GCN as a reference and proof of correct implementation. Similarly, forwards implementation shows identical performance and nearly identical training time compared to standard implementation, therefore is omitted except for GCN. Lastly, factorized GCN implementation refers to factorising the kernel arrow for GCN while still only using sender features. Also provided as a reference, it demonstrates training time overhead of "unnecessarily" factorising the kernel arrow and is yet another proof of implementation correctness.

We can see that both CatGNN and PyG implementations achieve identical test set accuracy on all layers, which indicates correct implementation. Although not explicitly shown, this also holds true for training and validation sets. Nevertheless, CatGNN is about 3 to 10 times slower than PyG depending on the type of layer used and the size of the dataset. Specifically, the difference between PyG and CatGNN seems to be the greatest for GATs where factorization of kernel arrow is required to pull receiver features. However, `GCN_2 (factored)` is not that much slower than `GCN_2` for all three datasets, which indicates that the sloweness comes mainly from the user implementation of GAT (specifically its attention coefficient calculation), rather than the base class implementation. This will be discussed later.

5.2 Graph classification

Following GNN literature and [FL19], we use standard graph classification datasets: Mutag, Proteins and Reddit-Binary [Mor+20]. We use three types of layers: GCN, GIN [Xu+18] and GraphSAGE [HYL17]. As before, all experiment setups are identical to [FL19] and we adapt PyG benchmarking code with small changes. Results are presented in Table 2. Average and standard deviation are caclulated over 10-fold cross validation.

Model	Mutag		Proteins		Reddit-Binary	
	Accuracy/%	Time/s	Accuracy/%	Time/s	Accuracy/%	Time/s
GCN_2	74.4 ± 9.9	0.047	70.8 ± 4.4	0.209	84.1 ± 2.7	1.159
PyG GCN	$\mathbf{74.0 \pm 9.8}$	**0.024**	$\mathbf{71.4 \pm 4.8}$	**0.116**	$\mathbf{85.1 \pm 4.4}$	**0.675**
GIN_2	84.1 ± 10.6	0.050	70.7 ± 4.4	0.217	89.7 ± 1.8	0.955
PyG GIN	$\mathbf{83.0 \pm 9.0}$	**0.023**	$\mathbf{71.0 \pm 4.3}$	**0.113**	$\mathbf{90.3 \pm 1.9}$	**0.629**
SAGE_2	75.5 ± 8.6	0.023	72.7 ± 4.1	0.109	88.6 ± 2.8	0.412
PyG SAGE	$\mathbf{74.5 \pm 10.1}$	**0.014**	$\mathbf{73.0 \pm 4.8}$	**0.067**	$\mathbf{89.4 \pm 2.6}$	**0.277**

Table 2: Graph classification. As before, accuracy columns refer to test accuracy. Time columns refer to the average time taken to execute a single training epoch.

Test accuracies are again identical given that the results are generally slightly unstable over multiple runs, and indicates correct implementation. Surprisingly, PyG is not that much faster than CatGNN, even for larger datasets like Reddit-Binary. In fact, PyG is at most 2 times faster than CatGNN. This is in contrast to semi-supervised node classfication tasks, where training time difference between CatGNN and PyG was larger for larger datasets like PubMed. One explanation could be mini-batching, as performance differences may not be that significant on smaller graphs.

5.3 Open Graph Benchmarks (OGB)

Finally, we evaluate CatGNN on Open Graph Benchmark (OGB) [Hu+20]. Only small OGB datasets are used due to limitations of Google Colab.

For semi-supervised node classification task, we use `ogbn-arxiv` dataset, a directed graph representing citation network and containing 169,343 nodes and 1,166,243 edges. Each node has 128 features and there are 40 classes. For benchmarking, we use Matthias Fey (OGB team) leaderboard code.[3] Since it only provides GCN and SAGE models, we also add GAT and SGC models with identical architectures. While this may not be the most efficient solution as indicated by the leaderboard ranking, at this point of development process we deem it to be more than enough for evaluation purposes. Results are presented in Table 3. Average and standard deviation are obtained over 10 runs due to resource constraints.

For graph classification task, we use molecular property prediction dataset `ogbn-molhiv` containing 41,127 graphs with approximately 25.5 nodes and 27.5 edges per graph, each node having 9 features and each graph belonging to one of two classes. For benchmarking, we use Weihua Hu (OGB team) leaderboard code.[4]. Due to time limitations, we only test GCN and GIN layers. Results of a single training run are presented in Table 4. Edge features are used in message construction but are never updated in line with the theory and the original benchmark code.

As before, test accuracies are identical for both libraries and datasets, indicating correct implementation. Evidently, however, PyG is much faster than CatGNN for large semi-supervised node classification datasets. Specifically for `ogbn-arxiv`, PyG is roughly 10 times faster for all types of layers. We also note that PyG is more memory efficient, as we occasionally ran into some memory consumption issues of CatGNN on the user implementation side and leave it for further work to address. On the other hand, graph classification results once again show that CatGNN is only around two times slower than PyG, even as datasets increase in size.

[3]Source code: https://github.com/snap-stanford/ogb/tree/master/examples/nodeproppred/arxiv
[4]Source code: https://github.com/snap-stanford/ogb/tree/master/examples/graphproppred/mol

Model	**ogbn-arxiv**	
	Test accuracy/%	Time/s
GCN_2	70.6 ± 0.3	1.99
GCN_2 (factored)	70.6 ± 0.3	2.21
GCN_2 (forwards)	70.5 ± 0.3	1.98
PyG GCN	$\mathbf{70.6 \pm 0.3}$	**0.18**
GAT_2	66.3 ± 0.4	1.20
PyG GAT	$\mathbf{65.6 \pm 0.8}$	**0.10**
SAGE_2	70.2 ± 0.3	2.38
PyG SAGE	$\mathbf{70.1 \pm 0.3}$	**0.20**
SGC_2	70.3 ± 0.3	3.20
PyG SGC	$\mathbf{70.3 \pm 0.3}$	**0.20**

Table 3: OGB semi-supervised node classification. As before, accuracy column refers to test accuracy. Time column refers to the average time taken to execute a single training epoch.

Model	**ogbn-mol**	
	Test ROC-AUC/%	Time/s
GCN_2	77.1	62.2
PyG GCN	**77.0**	**32.0**
GIN_2	75.6	59.8
PyG GIN	**75.0**	**38.5**

Table 4: OGB graph classification. ROC-AUC column refers to the test ROC-AUC of a single training run due to time constraints. Time column refers to the average time of a single training epoch execution.

5.4 Profiling integral transform implementation

Given empirical evaluation results, it is worthwhile to take a closer look at the bottlenecks of CatGNN backend and user implementations to identify potential areas for optimization. We present GCN and GAT layer training time profiles for `ogbn-arxiv` dataset. Specifically, we take one of the layers in the network and calculate the average time spent performing each of the four or five primitives in the user implementation. Results are shown in Figure 1.

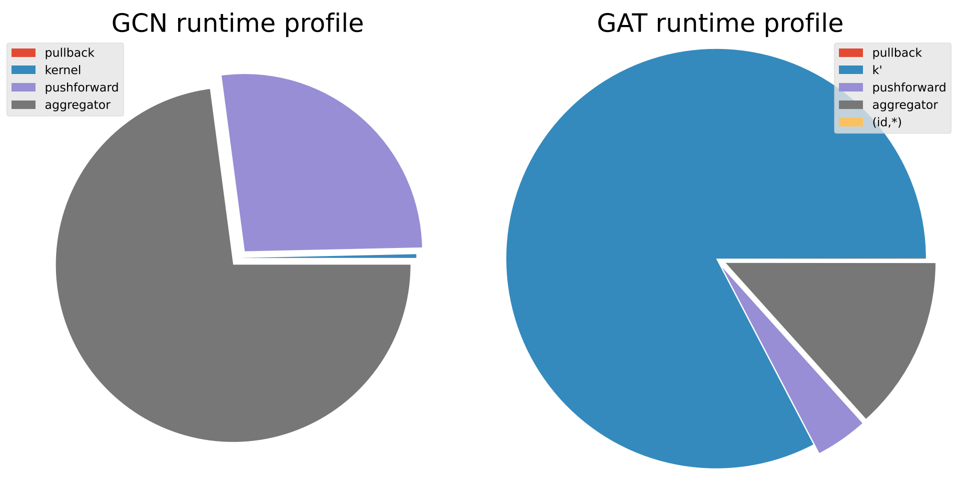

Figure 1: Pie chart of average time spent on each primitive for GCN and GAT user implementations on OGB arxiv dataset during training, in proportion to total time spent. The same arrows use the same colour in both charts.

For GCN, the costliest operation is the aggregator, taking roughly 75% of total time of all four arrows combined. Aggregator is followed by pushforward, while both pullback and kernel take up significantly less time. This is somewhat expected, since there is not much logic in either of these two arrows for GCN. On the other hand, GAT spends most of its time, roughly 80%, on kernel transformation, specifically k'. We still observe $\oplus$ taking roughly 3 times longer than t_*, however that is now insignificant compared to k'. Kernel being an expensive operation for GAT is also expected, since attention coefficients $\alpha_{i,j}$ have to be calculated within k'. On the other hand, $*$ is implemented rather efficiently in the base class. Therefore, long training times we observed for GAT could be improved by optimizing user implementation of factorized kernel transformation k'.

User implementation for pushforward in both GCN and GAT is the canonical pushforward that only calls t^{-1}, which is a base class implementation. Therefore, optimizing t^{-1} further is one of the key improvements we could make. However, aggregator in both GCN and GAT is purely user implementation that uses `pytorch_scatter`, meaning that optimizing aggregator falls on the user rather than the library.

6 Conclusion

In this mini-project report, we describe a prototype for a new category theory-based GNN library. We demonstrate implementations of well-known GNN layers in this new framework, a further proof of its correctness. We evaluate the performance of CatGNN by comparing it to leading "traditional" libraries like PyTorch Geometric on standard datasets. We are able to reproduce the results, indicating correct implementation. However, CatGNN is generally slower than PyG. We find that on semi-supervised node classification tasks, the difference in training time between PyG and CatGNN grows with the size of the dataset and depends on the layer type. This can be improved by optimizing both user implementation of the standard layers and library implementation of the utility functions $s, t, t^{-1}, *$. On the other hand, CatGNN was at most two times slower on graph classification tasks regardless of dataset size, demonstrating its feasibility for this type of tasks.

6.1 Future work

Majority of future effort should go into adjusting integral transform implementation based on received feedback. Following that, areas for optimization should be identified and more layers should be added, especially the ones supporting receiver features. Edge and graph-level feature update support should also be added once theory is extended.

More thorough benchmarking is required, for example investigation of GPU memory usage. Unfortunately, we have consistently ran into Colab problems with GPU memory (e.x. not resetting the memory when CUDA runs out of memory, less than 12GB available), which may be due to its free version.

One of the most interesting ideas that this framework allows us to explore is GNNs that choose their own spans (edges and/or nodes). CatGNN, specifically the standard (backwards) implementation, was designed with such task in mind as all utility functions (s, t, t^{-1}, f, $*$) support arbitrary choice of nodes/edges. Thus, such GNNs should be supported out-of-the-box given proper user implementation. Due to lack of time, however, this has not been explicitly explored in the mini-project.

Another interesting idea is GNNs that mask extra opposite edges. This was not a concern for GCNs and GATs: GCNs don't even use receiver features, while GATs always pull receiver features regardless of an opposite edge e^* existing in the graph or not. The framework and the API are powerful enough to support GNNs that, for example, only pull the receiver features when there exists an edge from receiver back to the sender. Masking option when retrieving opposite edges has been implemented in the base class, however has not been tested or further explored.

References

[VB21] Petar Veličković and Charles Blundell. "Neural algorithmic reasoning". In: *Patterns* 2.7 (July 2021), p. 100273. DOI: 10.1016/j.patter.2021.100273. URL: https://doi.org/10.1016%2Fj.patter.2021.100273.

[Xu+19] Keyulu Xu et al. *What Can Neural Networks Reason About?* 2019. DOI: 10.48550/ARXIV.1905.13211. URL: https://arxiv.org/abs/1905.13211.

[DV22] Andrew Dudzik and Petar Veličković. *Graph Neural Networks are Dynamic Programmers.* 2022. DOI: 10.48550/ARXIV.2203.15544. URL: https://arxiv.org/abs/2203.15544.

[KW16] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks.* 2016. DOI: 10.48550/ARXIV.1609.02907. URL: https://arxiv.org/abs/1609.02907.

[Vel+17] Petar Veličković et al. *Graph Attention Networks.* 2017. DOI: 10.48550/ARXIV.1710.10903. URL: https://arxiv.org/abs/1710.10903.

[FL19] Matthias Fey and Jan Eric Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ArXiv* abs/1903.02428 (2019).

[Hu+20] Weihua Hu et al. *Open Graph Benchmark: Datasets for Machine Learning on Graphs.* 2020. DOI: 10.48550/ARXIV.2005.00687. URL: https://arxiv.org/abs/2005.00687.

[Sen+08] Prithviraj Sen et al. "Collective Classification in Network Data". In: *AI Magazine* 29.3 (Sept. 2008), p. 93. DOI: 10.1609/aimag.v29i3.2157. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/2157.

[Wu+19] Felix Wu et al. *Simplifying Graph Convolutional Networks.* 2019. DOI: 10.48550/ARXIV.1902.07153. URL: https://arxiv.org/abs/1902.07153.

[Mor+20] Christopher Morris et al. "TUDataset: A collection of benchmark datasets for learning with graphs". In: *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020).* 2020. arXiv: 2007.08663. URL: www.graphlearning.io.

[Xu+18] Keyulu Xu et al. *How Powerful are Graph Neural Networks?* 2018. DOI: 10.48550/ARXIV.1810.00826. URL: https://arxiv.org/abs/1810.00826.

[HYL17] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs.* 2017. DOI: 10.48550/ARXIV.1706.02216. URL: https://arxiv.org/abs/1706.02216.

Appendix A: Vectorization

In this appendix, we explain how utility functions s, t, f, t^{-1} and $*$ are implemented in `BaseMPNNLayer_2` using vectorization. Vectorization is used to significantly speed-up extremely costly operations which would otherwise be implemented in Python using loops. Positive effects of such optimization can be seen by comparing performance of `BaseMPNNLayer_1` and `BaseMPNNLayer_2`. Note that V, E below are all arbitrary subsets of all nodes and edges of the graphs, allowing the user to implement GNNs that choose their own spans. Corresponding code can be found in `catgnn/integral_transform/mpnn_2.py`.

A.1 Senders and receivers of edges (s, t)

Implementations of $s : E \to V$ and $t : E \to V$, assuming sender-receiver order of E, are as follows:

```python
def s(self, E):
    return E[0]

def t(self, E):
    return E[1]
```

A.2 Features of vertices (f)

Implementation of $f : V \to R$ is as follows:

```python
def f(self, V):
    return self.X[V]
```

We assume that $\mathbf{X}$ is ordered by V, allowing us to quickly access features of all nodes of V at once.

A.3 Preimages of given vertices (t^{-1})

Efficient implementation of $t^{-1} : V \to P(E)$ for given vertices V is more tricky:

```python
def t_1(self, V):
    selected_E = self.E.T[torch.isin(self.t(self.E), V)].T
    return selected_E, self.t(selected_E)
```

We start by finding edges whose receivers are in the chosen vertices V by utilising `torch.isin()` method. This gives a boolean mask which we use to filter out edges from E which involve receivers not in V. Lastly, we return filtered edges and the receivers of these edges, which indicate the "bag" of values that the edge belongs to. This is by far the most efficient implementation we have tried out (e.x. compare to Python `lists` in `BaseMPNNLayer_1`), while still remaining decently close to paper's description.

A.4 Opposite edges ($*$)

For undirected graphs and for tasks where we do not care about opposite edges existing in the graph (e.x. GCN, GAT), finding opposite edges is as simple as swapping the rows of the edge matrix E:

```python
def get_opposite_edges(self, E, masking_required = False):
    return torch.flip(E, [0])
```

Otherwise, $*$ with masking is the trickiest operation to implement efficiently. Since some directed edges may not exist once the sender and receiver are flipped, we need to *mask* such edges. This can be achieved by converting them to a self-loop on a fake vertex, which is then assigned some standard feature value like `torch.nan` in $\mathbf{X}$:

```python
def get_opposite_edges(self, E, masking_required = True):
    flipped_E = torch.flip(E, [0])
    values, _ = torch.max((E == flipped_E.T.unsqueeze(-1)).all(dim=1).int(), dim=1)
    inverse_values = (values - 1) * (-1)
    return flipped_E.masked_fill(inverse_values, -1)
```

We still need to start by flipping the rows as before. We then need to check whether the flipped edges are indeed in E, this information is stored in `values`. Finally, we mask out the edges of `flipped_E` which are not in E by replacing them with a fake self-loop on a fake vertex of index `-1` (could be replaced by index $|E|$).

Appendix B: Example GCN user implementation

In this appendix, we give a simple example of how GCN can be implemented by the user using the standard (backwards) implementation. Corresponding code can be found in `catgnn/layers/gcn/gcn_mpnn_2.py`.

```python
from torch import nn
import torch_scatter

from catgnn.integral_transform.mpnn_2 import BaseMPNNLayer_2
from catgnn.utils import add_self_loops, get_degrees


class GCNLayer_MPNN_2(BaseMPNNLayer_2):
    def __init__(self, in_dim: int, out_dim: int):
        super().__init__()

        self.mlp_msg = nn.Linear(in_dim, out_dim, bias=False) # \psi

    def forward(self, V, E, X):
        # Add self-loops to the adjacency matrix
        E = add_self_loops(V, E)

        # Compute normalization as edge weights
        degrees = get_degrees(V, E)
        self.edge_weights = torch.sqrt(1 / (degrees[self.s(E)] * degrees[self.t(E)]))

        # Do integral transform
        return self.transform_backwards(V, E, X)

    def define_pullback(self, f):
        def pullback(E):
            return f(self.s(E)) # Canonical pullback
        return pullback

    def define_kernel(self, pullback):
        def kernel(E):
            # Message calculation
            return self.edge_weights.view(-1, 1) * self.mlp_msg(pullback(E))
        return kernel

    def define_pushforward(self, kernel):
        def pushforward(V):
            E, bag_indices = self.t_1(V)
            return kernel(E), bag_indices # Canonical pushforward
        return pushforward

    def define_aggregator(self, pushforward):
        def aggregator(V):
            edge_messages, bag_indices = pushforward(V)
            return torch_scatter.scatter_add(
                edge_messages.T, bag_indices.repeat(edge_messages.T.shape[0], 1)
            ).T[V] # Aggregator that adds the values in the bag of each node
        return aggregator

    def update(self, X, output):
        # This is either \phi or empty and \phi is put on top in the model like in PyG
        return output
```