

TF-GNN: Graph Neural Networks in TensorFlow

Oleksandr Ferludin^{*¶}, Arno Eigenwillig^{*¶}, Martin Blais^{*†}, Dustin Zelle^{*†}, Jan Pfeifer^{*¶}
 Alvaro Sanchez-Gonzalez^{*‡}, Sibon Li^{*‡}, Sami Abu-El-Haija[†], Peter Battaglia[‡], Neslihan Bulut[†]
 Jonathan Halcrow[†], Filipe Miguel Gonçalves de Almeida[†], Silvio Lattanzi[†], André Linhares[†]
 Brandon Mayer[†], Vahab Mirrokni[†], John Palowitch[†], Mihir Paradkar[¶], Jennifer She[‡]
 Anton Tsitsulin[†], Kevin Vilella[‡], Lisa Wang[‡], David Wong[‡], Bryan Perozzi^{*†}

¶: Google Core ML, †: Google Research, ‡: DeepMind
 tensorflow-gnn@googlegroups.com

Abstract

TensorFlow GNN (TF-GNN) is a scalable library for Graph Neural Networks in TensorFlow. It is designed from the bottom up to support the kinds of rich heterogeneous graph data that occurs in today’s information ecosystems. Many production models at Google use TF-GNN and it has been recently released as an open source project. In this paper, we describe the TF-GNN data model, its Keras modeling API, and relevant capabilities such as graph sampling, distributed training, and accelerator support.

1 Introduction

Machine Learning (ML) techniques have applications across domains as varied as medicine, social networks, biochemistry, robotics, and more. The success of many ML models is driven by their ability to incorporate different modalities of data (e.g. vision, text, sound, timeseries and geometric), each with its own unique structural (ir)regularities. Traditionally, software frameworks for machine learning (e.g. TensorFlow [1], PyTorch [29]) have focused on modeling one modality at a time, such as vision [23, 22] or natural language [27, 32, 10]. However, the development of graph representation learning [9] and subsequent industry interest [2, 15, 19, 6, 3, 30, 26, 37, 28] has motivated the need for better software frameworks for learning with *graph*-structured data.

In this paper we introduce TF-GNN², a Python framework that extends TensorFlow [1] with Graph Neural Networks (GNNs) [9, 17]: models that leverage graph-structured data. TF-GNN is motivated and informed by years of applying graph representation learning to practical problems at Google. In particular, TF-GNN focuses on the representation of *heterogeneous* graph data and supports the explicit modeling of an arbitrary number of relationships between an arbitrary number of entities (i.e., nodes). These relationships can be used in combination with other TensorFlow components, e.g., a TF-GNN model might connect representations from a language model to those of a vision model and fine-tune these features for a node classification task. Many teams at Google run TF-GNN models in production. We believe this to be a direct consequence of TF-GNN’s multi-layered API which is designed for accessibility to developers (regardless of their prior experience with machine learning).

Other software frameworks for learning from graph data have been proposed, most notably PyTorch Geometric (PyG) [11] and Deep Graph Library (DGL) [34]. We differ from DGL and PyG in three main ways. First, TF-GNN has been designed bottom-up for modeling heterogeneous graphs. Second, TF-GNN offers different levels of abstraction for increased modeling flexibility. Proficient users

^{*}TF-GNN Top contributors

²<https://github.com/tensorflow/gnn>

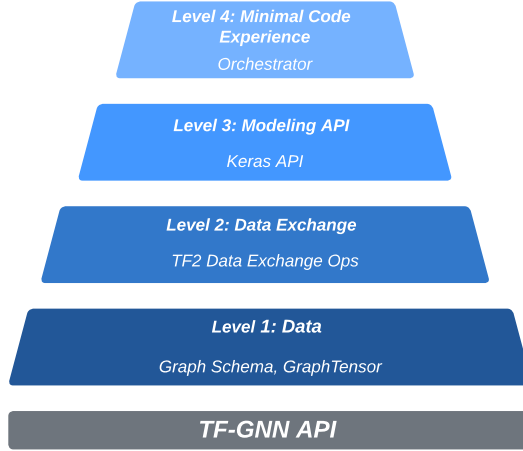


Figure 1: TF-GNN’s layered API decomposes the tools needed to create graph models into four distinct components of increasing abstraction.

can leverage raw TensorFlow operations for message passing, limited only by their imagination. Intermediate users use the Keras modeling API and pre-built convolution layers, while beginner users can use the Orchestrator to quickly experiment with GNNs. Finally, TF-GNN is programmed on top of TensorFlow. As such, its goal is to support the many production-ready capabilities present in the TensorFlow ecosystem.

Interestingly, these differences can provide advantages for several use cases. For instance, (i) while it is possible for PyG to model heterogeneous graph data, its syntax advocates partitioning a heterogeneous graph into a set of homogeneous graphs. This makes it (programmatically) challenging to create a graph layer that pools from multiple node features at once, or even create new node or feature types on the fly (i.e., through the network computation). However, TF-GNN’s flexibility allows for aggregating from different node or edge types at once. Furthermore, (ii) TF-GNN offers *edge-centric*, *node-centric*, and *graph-centric* building blocks for GNNs. Existing frameworks (e.g., PyG) are node-centric, making implementing some models more tedious. On the other hand, edge-centric models, such as Graph Transformers [35] can be natively expressed in TF-GNN. Finally, (iii) TF-GNN’s TensorFlow implementation inherits the benefits of the TensorFlow ecosystem, including access to model architectures for popular modalities (such as vision, text, and speech), and the ability to execute GNN models, for training and inference, on extremely fast hardware devices [18].

Summary of contributions. We present TF-GNN, an open-source Python library to create graph neural network models that can leverage heterogeneous relational data. TF-GNN enables training and inference of Graph Neural Networks (GNNs) on arbitrary graph-structured data. TF-GNN’s four API levels allow developers of all skill levels access to powerful GNN models. Many TF-GNN models run in production at Google. Finally, as a native citizen of the TensorFlow ecosystem, TF-GNN shares its benefits, including pretrained models for various various modalities (e.g., a NLP model) and support for fast mathematical hardware such as Tensor Processing Units (TPUs).

2 Overall design

TF-GNN offers a layered API to build Graph Neural Networks that lets users trade off flexibility for abstraction. From least to most abstract, the layers (shown in Fig. 1) are:

- **API Level 1:** The *Data Level* (§3) takes care of representing heterogeneous graphs and loading them into TensorFlow, including technicalities like batching and padding.
- **API Level 2:** The *Data Exchange Level* (§4.1) provides operations for sending information across the graph between its *nodes*, *edges* and the *graph context*.
- **API Level 3:** The *Model Level* (§4.2) facilitates writing trainable transformations of the data exchanged across the graph in order to update the state of nodes, edges and/or context.

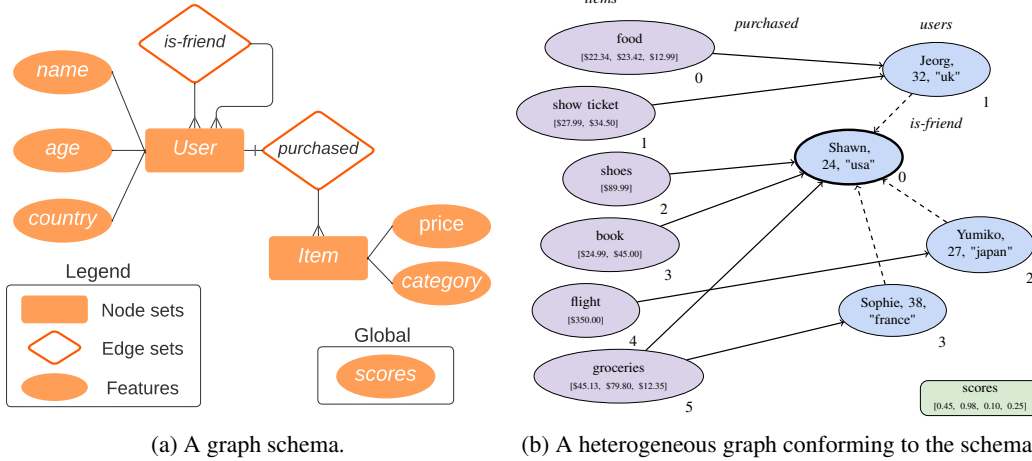


Figure 2: The recommending system example in the text uses the GraphSchema visualized in part (a). Part (b) shows a possible graph with the node sets, connecting edge sets, and features prescribed by the schema.

- **API Level 4:** The *Minimal-Code Experience Level* (§5) provides the Orchestrator, a toolkit for the easy composition of data input, feature processing, graph objectives, training, and validation.

This layered design is one reason for TF-GNN’s successful adoption for graph models at Google. Users can start at a high level and only later go in deeper to tweak parts of the model. Some users may choose to only use the data level and its associated tooling (like the graph sampler, §6.1) and use their own modeling framework. The following sections describe the API levels in greater detail. Finally, we discuss other parts of the library designed for use in production models (§6).

3 TF-GNN Heterogeneous Data Model (API Level 1)

To train a model on heterogeneous graph data, users first need to specify its node types, edge types and their respective features. This is done with the GraphSchema (§3.1). Based on that, the GraphTensor class (§3.2) can represent any graphs from the dataset.

3.1 Graph Schema

A GraphSchema object defines:

1. One or more named **node sets** and their respective features.
2. Zero or more named **edge sets** and their respective features. Each edge set has a specified source node set and a specified target node set. All edges in the set connect these node sets.
3. **Context features**, which pertain to the entire input graph.¹

As the name suggests, GraphSchema contains only an abstract definition of how entities are related (similar to an entity relational diagram [24]) and has no actual data points. By definition, the node sets are disjoint, so they can serve as node types; same for edges. For each feature, the graph schema defines its name, its datatype (int, float, or string) and its shape, as in TensorFlow [1]. That means the graph as a whole is heterogeneous, but within each node or edge set, the features are uniformly typed and shaped. Each feature can have a different shape, so the features of a node set might comprise a scalar (say, a categorical feature), a variable-length sequence (e.g., tokenized text), a fixed-length vector (such as a precomputed embedding), a rank-3 tensor with an RGB image, and so on.

An example schema for a prototypical recommendation systems problem is shown in Figure 2a. It defines a heterogeneous graph structure with the following structure:

¹Section 3.2 will refine the notion of context for graphs merged from a batch of inputs.

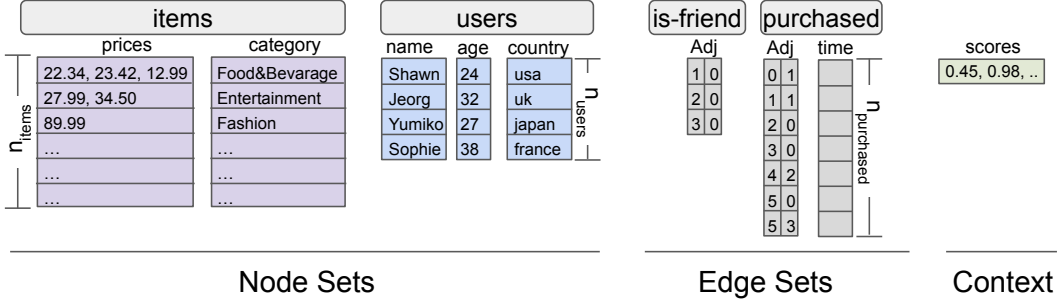


Figure 3: A GraphTensor to store the graph from Fig. 2b comprises tensors for all the features and for the adjacency data of each edge. (The size tensors are not shown here.) A Python expression for this GraphTensor object is shown in the Appendix.

- Two node sets: “items” and “users”. *Item* nodes have two features for their “category” (an integer scalar, corresponding to enumeration), and their “price” (a floating-point vector to hold item advertised prices). *Person* nodes have three features, representing their “name” (string), “age” (int), and “country” (int).
- Two distinct sets of edges: “purchased” and “is-friend”. *Purchased* edges connect users to items they have purchased, while *is-friend* edges connect users together.
- One context feature “scores”, which applies to the graph as a whole.

3.2 GraphTensor

Figure 2b shows a graph that conforms to the example schema from above: Each circle corresponds to a node (colored by its node set), and the two types of lines correspond to the two different edge sets. Each node contains the features specified for its node set.

Figure 3 shows our approach to representing one such a graph in TensorFlow. We index the nodes in each node set and the edges in each edge set as $0, 1, 2, \dots, n - 1$. Then any one feature on a node set or edge set can be represented by a tensor of shape $[n, f_1, \dots, f_k]$, where $[f_1, \dots, f_k]$, $k \geq 0$, is the feature shape from the GraphSchema. Moreover, for an edge set, the indices of source and target nodes are stored as two integer tensors of shape $[n]$ whose values are node indices in the node set specified by the graph schema.

The GraphTensor class expands on this approach to represent graphs as tensors through all stages of a TensorFlow program that builds a GNN model. Roughly, the stages are

1. Reading, shuffling, batching and parsing GraphTensor values from `tf.Example` records on disk; possibly distributing them between replicas for data-parallel training.
2. Transforming one or more input features per node or edge into a fixed-size representation for deep learning.
3. Running a graph neural network for several rounds to update the hidden states of nodes (and possibly edges) from neighboring parts of the graph, followed by reading out the relevant hidden states and computing the model’s output.

GraphTensor supports batching natively, and is indeed a tensor of graphs with a shape $[g_1, \dots, g_r]$. A scalar GraphTensor has shape $[]$ and holds a single graph, while a GraphTensor of shape $[g_1]$ holds a vector of g_1 graphs, as usual for training with minibatches. Ranks $r > 1$ are rarely needed.

Each node set and edge set holds a dictionary of named features. Each feature is a tensor of shape $[g_1, \dots, g_r, n, f_1, \dots, f_k]$. GraphTensor allows that a feature dimension f_i may vary between the items of one node/edge set, or that the number n of items varies between the multiple graphs in a GraphTensor. In both cases, the solution is to store the feature as a `tf.RaggedTensor`, not a `tf.Tensor`. Under the hood, this stores an explicit partitioning for each non-uniform, or “ragged”, dimension. The shape reports its size as `None`.

To support feature processing, TF-GNN lets you create a new `GraphTensor` from an old one by replacing some or all of the features while keeping track of the implied schema change.

Finally, in service of training models, `GraphTensor` provides a method to merge a batch of inputs to a scalar `GraphTensor`. For each node/set, this method concatenates its elements across the batch of inputs and adjusts the node indices stored on edges correspondingly. The result is a `GraphTensor` of shape $[]$ with a flat index space $0, 1, \dots, n_{\text{total}} - 1$ for each node/edge set, across the boundaries of batched examples. Features get the shape $[n_{\text{total}}, f_1, \dots, f_k]$. Conveniently, such features can be represented as a `tf.Tensor` if all feature dimensions are fixed, even if a node/edge set's size n_{total} is not constant between batches. If necessary, it can be made constant as well by adding a suitably sized padding graph to each batch of input graphs and assigning it weight 0 for training the GNN. Standard GNN operations on nodes and edges respect the boundaries between the merged input graphs, because there are no edges connecting them. To achieve the same for context features, `GraphTensor` supports the notion of **components** in a graph, and stores context features indexed by component.

4 Modeling with TF-GNN

The core of TF-GNN is specifying how a computation utilizes graph structured data. In this section we detail the low level (TensorFlow) and high level (Keras) APIs used to construct GNNs.

4.1 Data Exchange Ops (API Level 2)

TF-GNN sends data across the graph as follows. *Broadcasting* from a node set to an edge set returns for each edge the value from the specified endpoint (say, its source node). *Pooling* from an edge set to a node set returns for each node the specified aggregation (sum, mean, max, etc.) of the value on edges that have the node as the specified endpoint (say, their target node.) The tensors involved are shaped like features of the respective node/edge set in the `GraphTensor` and can, but need not, be stored in it. Similarly, graph context values can be broadcast to or pooled from the nodes or edges of each graph component in a particular a node set or edge set.

Unlike multiplication with an adjacency matrix, this approach provides a natural place to insert per-edge computations with one or more values, such as computing attention weights [33, 7], integrating edge features into messages between nodes [12], or maintaining hidden states on edges [5].

4.2 Model Building API (API Level 3)

At API Level 3, TF-GNN follows standard TensorFlow practice and adopts Keras to express trainable transformations and their composition into models. API Levels 1 and 2 can serve other ways of modeling just as well. The shape $[n, \dots]$ of feature tensors allows to reuse standard neural network layers for item-wise transformations of node/edge sets, with set size n in place of a batch size.

A typical GNN model (cf. §3.2) consists of (i) feature transformations, (ii) a GNN core, and (iii) the final readout and prediction. TF-GNN lets you express this as a sequence of Keras layers that each take a `GraphTensor` input and return a `GraphTensor` output with transformed features – or a `Tensor` for reading out the final prediction.

4.2.1 Feature transformation layers

The feature transformations treat each node/edge set in isolation. Depending on the available features, they can range from simple numeric transformations to running an entire deep learning model to compute an embedding of, say, image or text data. TF-GNN lets you plug in other TensorFlow models and fine-tune them jointly while training the GNN on top. In the end, the representations of multiple input features are combined to form one initial **"hidden_state"** feature. TF-GNN's `MapFeatures` layer makes it easy to build Keras sub-models for each node/edge set that map a features dict to a transformed features dict, and eventually the hidden state.

4.2.2 Graph Neural Network layers

The GNN core of the model is expressed as a sequence of *graph update* layers, each of which accepts a GraphTensor with "hidden_state" features and returns a new GraphTensor with these features updated. Each layer object has its own trainable weights; weight sharing is achieved by using the same layer object repeatedly.

Users can define their own graph update layers, or reuse those from the growing collection of models bundled with the TF-GNN library. A user-defined graph update can, if needed, contain free-form code with an arbitrary composition of trainable transformations and broadcast/pool operations across all parts of the graph.

More commonly, graph updates are constructed from pieces operating on individual node sets or edge sets. TF-GNN provides a generic GraphUpdate class for that purpose, based on the following breakdown of updating a heterogeneous graph.

Consider any node v in some node set V of a heterogeneous graph. Starting from the initial hidden state $\mathbf{h}_v^{(0)}$, successive GraphUpdates compute the hidden state of v as

$$\mathbf{h}_v^{(i+1)} = \text{NEXTNODESTATE}_V^{(i+1)}(\mathbf{h}_v^{(i)}, \overline{\mathbf{m}}_{E_1,v}^{(i+1)}, \dots, \overline{\mathbf{m}}_{E_k,v}^{(i+1)}) \quad (1)$$

using the pooled messages $\overline{\mathbf{m}}_{E_j,v}^{(i+1)}$ received by node v in round $i+1$ along all edge sets E_1, \dots, E_k incident to node set V . Notice that their number k is a constant from the graph schema for all $v \in V$.

Let $\mathcal{N}_{E_j}(v) = \{u \mid (u, v) \in E_j\}$ denote the neighbors of v along one edge set E_j , and notice that the size of $\mathcal{N}_{E_j}(v)$ may vary with $v \in V$. GraphUpdate supports two ways of computing $\overline{\mathbf{m}}_{E_j}$: in one step directly from the neighbor nodes as

$$\overline{\mathbf{m}}_{E_j,v}^{(i+1)} = \text{CONV}_{E_j}^{(i+1)}(\mathbf{h}_v^{(i)}, \{\mathbf{h}_u^{(i)} \mid u \in \mathcal{N}_{E_j}(v)\}), \quad (2)$$

or in two steps, materializing a per-edge message in the GraphTensor as

$$\begin{aligned} \mathbf{m}_{E_j,(u,v)}^{(i+1)} &= \text{NEXTEDGESTATE}_{E_j}^{(i+1)}(\mathbf{h}_u^{(i)}, \mathbf{h}_v^{(i)}, \mathbf{m}_{E_j,(u,v)}^{(i)}), \\ \overline{\mathbf{m}}_{E_j,v}^{(i+1)} &= \text{EDGEPOOL}_{E_j}^{(i+1)}(\mathbf{h}_v^{(i)}, \{\mathbf{m}_{E_j,(u,v)}^{(i+1)} \mid u \in \mathcal{N}_{E_j}(v)\}). \end{aligned} \quad (3)$$

The two-step approach of Eq. (3) supports recurrence in $\mathbf{m}_{E_j,(u,v)}$, effectively turning it into a hidden state for edges. With that, and a context (or "global") state not shown here, this approach covers Graph Networks [5] and generalizes them to heterogeneous graphs.

Without recurrence in $\mathbf{m}_{E_j,(u,v)}$ (but possibly a constant edge feature in its place) this approach also covers Message Passing Neural Networks [12], generalized to heterogeneous graphs. The CONV abstraction from Eq. (2) is useful to express any of a number of successful GNN architectures in a single Python class and to transfer them directly to heterogeneous graphs with an arbitrary schema. Section 4.3 and the Appendix review some concrete cases, including Graph Convolutional Networks [21], which popularized the term *graph convolution* that we adopt here.

4.3 Implementing Popular Architectures

Graph Convolutional Networks [21]: GCNs for homogeneous graphs rely on adding loops (v, v) to the single edge set E to feed $\mathbf{h}_v^{(i)}$ into the computation of $\mathbf{h}_v^{(i+1)}$ along with the neighbor nodes. TF-GNN implements them by specializing Eq. (1) and (2) to

$$\mathbf{h}_v^{(i+1)} = \overline{\mathbf{m}}_v^{(i+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{\sqrt{d_u d_v}} \mathbf{W}^{(i+1)} \mathbf{h}_u^{(i)}\right), \quad (4)$$

where d_u is the in-degree of node u including loops and σ is an activation function, such as ReLU. Observe how CONV from Eq. (2) is the graph convolution, and NEXTNODESTATE trivially passes through its result.

The relational extension, **R-GCN**, [31] considers heterogeneous graphs and uses

$$\begin{aligned} \mathbf{h}_v^{(i+1)} &= \sigma\left(\sum_{j=1}^k \overline{\mathbf{m}}_{E_j,v}^{(i+1)} + \mathbf{W}_V^{(i+1)} \mathbf{h}_v^{(i)}\right), \\ \overline{\mathbf{m}}_{E_j,v}^{(i+1)} &= \frac{1}{|\mathcal{N}_{E_j}(v)|} \sum_{u \in \mathcal{N}_{E_j}(v)} \mathbf{W}_{E_j}^{(i+1)} \mathbf{h}_u^{(i)} \end{aligned} \quad (5)$$

with separate weights for each edge set and node set. This translates immediately to CONV and NEXTNODESTATE maps.

GraphSAGE [16] considers homogeneous sampled subgraphs and proposes several aggregator architectures, which translate directly to choices for CONV in Eq. (2). Its NEXTNODESTATE function turns out to be the special case $k = 1$ of Eq. (5) for R-GCN. In the GraphUpdate framework, GraphSAGE generalizes naturally to the heterogeneous case by running its CONVs on multiple edge sets and combining them as in Eq. (5) for $k > 1$.

GAT [33] extends GCN by replacing the weighted sum in Eq. (4) by a concatenation of multiple weighted sums (attention heads), each with its own data-dependent weighting. Both GAT and its modification **GATv2** [7] can be expressed as a CONV operation (formulas omitted for brevity).

The GraphUpdate framework allows to generalize GAT/GATv2 directly to the heterogeneous case, with no extra coding, analogous to the generalization from GCN to R-GCN. Attention is distributed separately between the edges of each edge set; learning the relative importance of different edge sets (relation types) is left to their separate weight matrices $\mathbf{W}_{E_j}^{(i+1)}$ in Eq. (5).

TF-GNN provides a base class for implementing CONV operations that allows a unified implementation of attention (i) from a node onto its neighbors, possibly combining the neighbor node state with a feature from the connecting edge; (ii) from a node onto its incoming edges; (iii) from the graph context onto all nodes; (iv) from the graph context onto all edges. Cases (ii–iv) provide attention for all aggregation steps of Graph Networks [5]. The appendix shows the unified implementation of GATv2 attention for all four cases.

5 Orchestration (API Level 4)

At the highest level, TF-GNN provides the *Orchestrator*: a quick-start toolkit with solutions for common graph learning tasks. It includes popular graph learning objectives, distributed training capabilities, accelerator support and the handling of numerous TensorFlow idiosyncrasies. The Orchestrator aims to collect the tools necessary for (i) elevating the novice to a TF-GNN power user and (ii) increasing the scope of the graph learning expert’s innovation. The toolkit supports graph learning research by offering both a standard framework for the reproduction of results and a shared catalog of SotA and convenience graph learning techniques and objectives.

Specifically, the Orchestrator performs the following steps:

1. Reading input data to extract input graph(s) as `GraphTensor` instances \mathbf{X} and corresponding label(s) \mathbf{Y} .
2. Processing features for a specific dataset $\mathbf{X} \rightarrow \mathbf{X}'$
3. Adapting a model to the graph learning objective $\mathbf{M} \rightarrow \mathbf{M}'$.
4. Training the adapted model $\mathbf{M}': \mathbf{X}' \rightarrow \mathbf{H}$ to minimize the loss between \mathbf{H} and \mathbf{Y} .
5. Exporting the model (\mathbf{M}') for inference or deployment.

The Orchestrator provides abstractions for the composition of these five steps as follows.

- **Data source** (Python protocol `DatasetProvider`): An arbitrary source (e.g., files on disk) that produces `GraphTensor` and corresponding schema.
- **Feature processing** (Python Protocol `GraphTensorProcessorFn`): Feature manipulations for a `GraphTensor`—typically associated with a specific dataset.
- **Task** (Python Protocol `Task`): A collection of the ancillary pieces for a graph learning objective. Provides mechanisms for extending an arbitrary model to a graph learning objective (e.g., node classification or regression) and processing for the data source manipulations of that same graph learning objective.
- **Model**: An arbitrary model that operates on `GraphTensor`. The model is adapted to a graph learning objective by the above **Task**.
- **Training Hyperparameters**: Including the choice of optimization algorithm (e.g., ADAM [4]), learning rate, as well as model-hyperparameters (e.g., number of layers and their widths). Integrated with a automated hyperparameter tuning service [13].

6 Sampling and Scaling

At Google, we need to build neural network models for graphs of incredible size (trillions of edges). Heterogeneous graphs of this scale that have rich node and edge features cannot fit in the memory of single machines. Furthermore, for fast training and inference, deep graph models must be able to exploit parallel computations on specialized hardware.

6.1 Distributed Sampling for Training and Inference

On massive, well-connected graphs, even deep GNNs rarely aggregate features from distant nodes. As a result, when making a prediction on a root node (or a root activation node for edge/neighborhood prediction), it is nearly-equivalent for the GNN to operate on a sufficiently-wide *subgraph* around the root. For example, training and inference with a 2-layer GCN [21] only requires the 2-hop subgraph around the root.

To train GNNs on massive graphs, we first run a distributed *subgraph sampling* operation [16] which queries each root node for a subgraph, and writes each subgraph to disk as an individual GraphTensor. The subgraph GraphTensors are (randomly) grouped into file shards, allowing distributed training using TensorFlow. Once the model is trained, inference on large graphs is done identically by sharding the graph into rooted subgraphs at each inference node. See Figure 4 for an illustration.

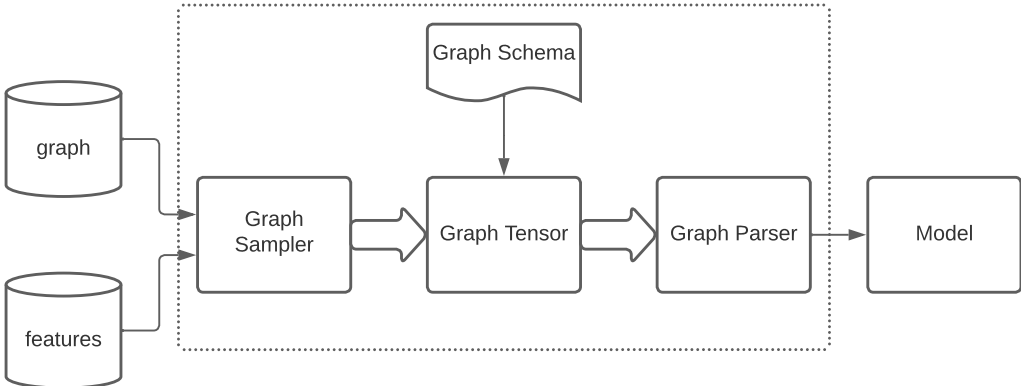


Figure 4: Diagram of massive-graph sampling and training pipeline with TF-GNN.

6.2 In-memory Learning

In settings where the input graphs are small enough to fit into memory (e.g., of the GPU), then it might be possible to perform entire-graph learning: where all graph nodes, edges, and features, are loaded for processing at once. Here, one can express the objective function on the entire graph, e.g., as a node classification cross-entropy loss on all labeled nodes. TF-GNN offers functionality and tutorials for these settings.¹

7 Related Work

Here we provide an overview of related work, which we separate into two broad categories: single-machine and distributed software frameworks. While it is more common for single-machine frameworks to be on the limits of research and innovation, industrial applications require distributed support and pose challenges rarely addressed by pure research applications.

Single-machine libraries. PyG (PyTorch-Geometric) [11] is a de-facto standard framework for GNNs in the PyTorch ecosystem. PyG provides automatic batching support, GPU acceleration, and an interface to common graph learning datasets. Its performance is further enhanced by a set

¹https://github.com/tensorflow/gnn/tree/main/examples/in_memory

of optimized sparse GPU kernels tailored towards graph learning workloads. Spektral [14] is a prominent TensorFlow framework that follows Keras model building principles. It offers a similar experience to PyG in the TensorFlow ecosystem but without any batching support.

TF-GNN differs from these in many ways. For example, Spektral computes edge-centric functions (such as, GAT) on minibatches by converting sparse adjacency matrices to dense ones. In general, these frameworks are not designed to scale to large graphs, but allow for easy experimentation by researchers on small graphs. On the other hand, TF-GNN’s primary purpose is to scale to large graphs.

Distributed libraries. Deep Graph Library (DGL) [34] allows switching the backend platform between PyTorch, TensorFlow, and Apache MXnet with minimal code modifications. DistDGL expension [36] enables efficient multi-machine training with DGL. Training is supposed to be done on a fleet of CPU-heavy instances connected in a cluster with a wide communication channel. DistDGL is mainly focused on reducing communication between the workers, each of which is performing sampling and training simultaneously. It partitions the input graph with METIS [20] and uses each partition as an example. Unlike DistDGL, TF-GNN’s distributed training is more general. TF-GNN does not assume that the data contains clusters, or that the graph structure fit into memory for partitioning via METIS.

Graph-Learn (formerly AliGraph) [38] is an open-source industrial graph learning framework built on top of TensorFlow. It is designed to natively handle large heterogeneous graphs, and it employs several techniques to facilitate large-scale training. Their distribution strategy relies on distributing the graph among workers machines, with a requirement that all worker machines must be alive at the same time: their training would stop if any worker machine fails. This differs from the distribution strategy of TF-GNN. In particular, TF-GNN samples a large graph into subgraphs using a resilient distributed system [8]. Similarly, TF-GNN can be used with the asynchronous distributed model training in TensorFlow, which is robust to machine failures.

Paddle Graph Learning (PGL) [25] is probably the most similar to TF-GNN. It is founded upon message passing over heterogeneous graphs. There are two notable differences between PGL and TF-GNN. First, PGL is more restrictive: each node must have a single feature (it is non-trivial to combine visual feature and textual feature, per node, per se) and dictates that all nodes must have the same feature dimensions. In contrast, TF-GNN support multiple features per node type (including ragged feature dimensions) and two node types can hold different features. Second, PGL uses Paddle as the computation backend whereas TF-GNN uses TensorFlow. From a practical sense, it is easier to find state-of-the-art (SotA) network architectures and pre-trained models in TensorFlow, which would make it easier to combine per-modality SotA models within a GNN.

8 Conclusions

In this work we have presented TF-GNN, our open source framework used for Graph Neural Networks at Google. TF-GNN is a software framework which reduces the technical burden for GNN productization and facilitates experimentation with GNNs. TF-GNN’s expressive modeling capability allows complex relationships between nodes, edges, and graph-level elements in a model. This enabling the straight-forward implementation of intricate models. TF-GNN offers four levels of increasingly abstract APIs, serving a range of use-cases from beginners with a graph problem but little ML experience to ML researchers who desire complete control of their graph learning system. Many models at Google already use TF-GNN, and we believe that this project will accelerate the industrial adaptation of these promising models at more organizations.

References

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.
- [2] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. Mixhop: Higher-order graph convolutional

- architectures via sparsified neighborhood mixing. In *international conference on machine learning*, pages 21–29. PMLR, 2019.
- [3] Rami Al-Rfou, Bryan Perozzi, and Dustin Zelle. Ddgc: Learning graph representations for deep divergence graph kernels. In *The World Wide Web Conference*, pages 37–48, 2019.
 - [4] Jimmy Ba and Diederik Kingma. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.
 - [5] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv/1806.01261*, 2018.
 - [6] Aleksandar Bojchevski, Johannes Klicpera, Bryan Perozzi, Amol Kapoor, Martin Blais, Benedek Rózemberczki, Michal Lukasik, and Stephan Günnemann. Scaling graph neural networks with approximate pagerank. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2464–2473, 2020.
 - [7] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? In *International Conference on Learning Representations*, 2022.
 - [8] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. *ACM Sigplan Notices*, 45(6):363–375, 2010.
 - [9] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Re, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy. *Journal of Machine Learning Research*, 23(89):1–64, 2022.
 - [10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, pages 4171–4186. Association for Computational Linguistics, 2019.
 - [11] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
 - [12] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, 2017.
 - [13] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1487–1495, 2017.
 - [14] Daniele Grattarola and Cesare Alippi. Graph neural networks in tensorflow and keras with spektral (application notes). *IEEE Computational Intelligence Magazine*, 16(1):99–106, 2021.
 - [15] Jonathan Halcrow, Alexandru Mosoi, Sam Ruth, and Bryan Perozzi. Grale: Designing networks for graph learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD ’20, page 2523–2532, New York, NY, USA, 2020. Association for Computing Machinery.
 - [16] W. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 2017.
 - [17] William L. Hamilton. *Graph Representation Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2020.

- [18] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.
- [19] Amol Kapoor, Xue Ben, Luyang Liu, Bryan Perozzi, Matt Barnes, Martin Blais, and Shawn O'Banion. Examining covid-19 forecasting using spatio-temporal graph neural networks. *arXiv preprint arXiv:2007.03113*, 2020.
- [20] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.
- [21] Thomas Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.
- [22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Proc. of the Advances in Neural Information Processing Systems (NIPS)*, 2012.
- [23] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.
- [24] Qing Li and Yu-Liu Chen. Entity-relationship diagram. In *Modeling and Analysis of Enterprise and Information Systems*, pages 125–139. Springer, 2009.
- [25] Yanjun Ma, Dianhai Yu, Tian Wu, and Haifeng Wang. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Computing*, 1(1):105–115, 2019.
- [26] Elan Sopher Markowitz, Keshav Balasubramanian, Mehrnoosh Mirtaheri, Sami Abu-El-Haija, Bryan Perozzi, Greg Ver Steeg, and Aram Galstyan. Graph traversal with tensor functionals: A meta-algorithm for scalable learning. In *International Conference on Learning Representations*, 2021.
- [27] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*, 2013.
- [28] John Palowitch, Anton Tsitsulin, Brandon Mayer, and Bryan Perozzi. Graphworld: Fake graphs bring real insights for gnns. *arXiv preprint arXiv:2203.00112*, 2022.
- [29] Adam Paszke, Sam Gross, Francisco Massa, and Others. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.
- [30] Benedek Rozemberczki, Peter Englert, Amol Kapoor, Martin Blais, and Bryan Perozzi. Pathfinder discovery networks for neural message passing. In *Proceedings of the Web Conference 2021*, pages 2547–2558, 2021.
- [31] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with Graph Convolutional Networks. In Aldo Gangemi et al., editors, *Proc. ESWC*, volume 10843 of *LNCS*, pages 593–607. Springer, 2018.
- [32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.
- [33] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.
- [34] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

- [35] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. Graph transformer networks. In *Advances in Neural Information Processing Systems*, 2019.
- [36] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.
- [37] Qi Zhu, Natalia Ponomareva, Jiawei Han, and Bryan Perozzi. Shift-robust gnns: Overcoming the limitations of localized graph training data. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 27965–27977. Curran Associates, Inc., 2021.
- [38] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019.

A Appendix

A.1 Graph Tensor Example

Consider again the example schema and graph provided in Figures 2 and 3. This example graph would be represented with the following tensors:

- Node feature tensors:
 - “items” category, string vector tensor: `["food", "show ticket", "shoes", "book", "flight", "groceries"]`, with shape `[6]`
 - “item” price, float tensor: `[[22.34, 23.42, 12.99], [27.99, 34.50], [89.99], [24.99, 45.00], [350.00], [45.13, 79.80, 12.35]]`, with shape `[6, None]`; an instance of `tf.RaggedTensor`
 - “users” name, string vector tensor: `["Shawn", "Jeorg", "Yumiko", "Sophie"]`, with shape `[4]`
 - “users” age, int vector tensor: `[24, 32, 27, 38]`, with shape `[4]`
 - “users” country: int vector tensor: `[3, 2, 1, 0]`, with shape `[4]`, assuming that the country vocabulary enumeration is `dict(france=0, japan=1, uk=2, usa=3)`.
- Edge feature tensors:
 - “purchased” with
 - * source indices, int vector tensor: `[0, 1, 2, 3, 4, 5, 5]`.
 - * target indices, int vector tensor: `[1, 1, 0, 0, 2, 3, 0]`, both with shape `[7]`.
 - “is-friend” with
 - * source indices, int vector tensor: `[1, 2, 3]`.
 - * target indices, int vector tensor: `[0, 0, 0]`, both with shape `[3]`.
- Graph-level feature tensors:
 - “scores”, float matrix tensor: `[[0.45, 0.98, 0.10, 0.25]]` with shape `[1, 4]`.

Note how the edges connectivity is encoded as source and target indices in the arrays of node features. The indices are indicating, for each edge, which position in the node feature array they are referring to. For example, the fifth values of the ‘purchased/#source’ and ‘purchased/#target’ is `[4, 2]`, which link together nodes “flight” and “Yumiko”.

A.1.1 GraphTensor Features & Shapes

Shapes of feature tensors, stored in `GraphTensor`, are crucial for building models in TF-GNN. Each tensor has a particular shape constraint based on its containing node set, edge set, or context dictionary. The shapes are as follows:

- Node features. All the features associated with a node set share the initial dimension, which is the total number of nodes in the node set. In the example above, features for node “items” have 6 nodes, and so all their tensor shapes are of the form `[6, ...]`. In general, node features will have the `[num_nodes, feature...]` shape. For example, a simple 64-dimensional embedding of each node results in shape `[6, 64]`, while a 224x224 image with 3 color channels stored at each node could result in `[6, 224, 224, 3]`.
- Edge features and indices. Similarly, all the features associated with an edge set shares the leading dimension, which is the total number of edges in the edge set. This includes the edge indices (rendered above as special features `source` and `target`). In the example, features for edges “purchased”, both of these have shape `[7]`. If edges have information encoded as features, *e.g.*, an embedding of shape `[32]`, then the edge feature tensor would be of shape `[7, 32]`. In general, edge features will have shape `[num_edges, feature...]`.
- Context features. Context features apply to one component of the graph. An input graph parsed with a `GraphSchema` has a single component, so its context features have shapes `[1, ...]`. After batching inputs and merging batches of inputs into components of a single graph (see §3.2), there are multiple components, and their number appears as the outermost dimension in the shape of context features.

A.2 Graph Tensor API

A.2.1 GraphTensor Interface

The GraphTensor objects you obtain from the parser are lightweight containers for all the dense and ragged features that are part of an example graph, as well as the adjacency information. These can contain a single example graph or a batch of multiple graphs. You can access the tensors with an interface similar to that of Python dicts. For example, to access the age feature in the example, you would do this:

```
graph.node_sets["users"]["age"]  
  
<tf.Tensor: shape=(4,), dtype=int32, numpy=array([24, 32, 27, 38],  
          dtype=int32)>
```

Edge indices are accessed with their "adjacency" property:

```
graph.edge_sets["purchased"].adjacency.source  
  
<tf.Tensor: shape=(7,), dtype=int32, numpy=array([0, 1, 2, 3, 4, 5, 5],  
          dtype=int32)>
```

And similarly for context features:

```
graph.context["scores"]  
  
<tf.Tensor: shape=(1, 4), dtype=float32, numpy=array([[0.45, 0.98, 0.1  
          , 0.25]], dtype=float32)>
```

A.2.2 Creating GraphTensors

Instances of GraphTensor can also be created as constants or from existing tensors. This is useful for writing unit tests and working in a Colab. To create a GraphTensor, you have to provide the various pieces and features forming the GraphTensor. This is best described by an example:

```
graph = tfgnn.GraphTensor.from_pieces(  
    context=tfgnn.Context.from_fields(  
        features={  
            "scores": [[0.45, 0.98, 0.10, 0.25]],  
        }  
    ),  
    node_sets={  
        "items": tfgnn.NodeSet.from_fields(  
            sizes=[6],  
            features={  
                "category": ["food", "show ticket", "shoes",  
                             "book", "flight", "groceries"],  
                "price": tf.ragged.constant([[22.34, 23.42, 12.99],  
                                             [27.99, 34.50],  
                                             [89.99],  
                                             [24.99, 45.00],  
                                             [350.00],  
                                             [45.13, 79.80, 12.35]]),  
            }  
        ),  
        "users": tfgnn.NodeSet.from_fields(  
            sizes=[4],  
            features={  
                "name": ["Shawn", "Jeorg", "Yumiko", "Sophie"],  
                "age": [24, 32, 27, 38],  
                "country": ["usa", "uk", "japan", "france"],  
            }  
        ),  
    },  
    edge_sets={  
        "purchased": tfgnn.EdgeSet.from_fields(  

```

```

        sizes=[7],
        features={},
        adjacency=tfgnn.Adjacency.from_indices(
            source=("items", [0, 1, 2, 3, 4, 5, 5]),
            target=("users", [1, 1, 0, 0, 2, 3, 0]),
        ),
        "is-friend": tfgnn.EdgeSet.from_fields(
            sizes=[3],
            features={},
            adjacency=tfgnn.Adjacency.from_indices(
                source=("users", [1, 2, 3]),
                target=("users", [0, 0, 0]),
            ),
        ),
    })
})

```

The data types for `tfgnn.Context`, `tfgnn.NodeSet`, `tfgnn.EdgeSet`, and `tfgnn.Adjacency` are exposed to the API for this construction to be possible. The classes verify that the ranks and shapes of the tensors are matching each other as constrained by the graph structure (i.e., in the same set they share a common prefix dimensions).

A.2.3 Reading GraphTensors

Graphs can be written to disk by encoding them into streams of `tf.train.Example` protos.

Graphs can be read into a TensorFlow program by decoding these `Example` protos to `GraphTensor` objects. To configure the parsing routine, the `GraphSchema` message is read and converted to a `GraphTensorSpec` object which describes the layout of the graph tensor in the TensorFlow runtime: the list of its various node sets, edge sets, and all the features associated with them and the context features. This data structure is analogous to the `tf.TensorSpec` object of TensorFlow. A `GraphTensorSpec` object is attached to all instances of `GraphTensor` and flows along with it. To read a stream of graph tensors, first create a spec, like this:

```

import tensorflow_gnn as tfgnn

graph_schema = tfgnn.read_schema(schema_path)
graph_spec = tfgnn.create_graph_spec_from_schema_pb(graph_schema)

```

You can then use the library's own parser to decode the `tf.train.Example` features into `GraphTensor` objects:

```

data_paths = tf.data.Dataset.list_files(...)
ds = tf.data.TFRecordDataset(data_paths)
ds = ds.batch(batch_size)
parser_fn = functools.partial(tfgnn.parse_example, graph_spec)
ds = ds.map(parser_fn)

```

The dataset `ds` yields a stream of instances of `GraphTensor`. The parsing function ingests a batch of serialized graphs (a rank-1 tensor of strings).

Similarly to `tf.io.parse_single_example`, there is also a function `tfgnn.parse_single_example` to parse an unbatched stream of encoded `tf.train.Example` strings with `GraphTensor` instance in them, and you can also run batch on top of that (i.e., you can batch `GraphTensor` instances):

```

data_paths = tf.data.Dataset.list_files(...)
ds = tf.data.TFRecordDataset(data_paths)
parser_fn = functools.partial(tfgnn.parse_single_example, graph_spec)
ds = ds.map(parser_fn)
ds = ds.batch(batch_size)

```

Typically this step is followed by a feature engineering step that normalizes, embeds and concatenates the various features into a single embedding for each set, and then batches multiple subgraphs into modified batched `GraphTensor` instances.

The containers can then be used to pick out various tensors and build a model, or simply to inspect their contents:

```
for graph in ds.take(10):
    tensor = graph.node_sets["item"]["category"]
    print(tensor)
```

You will go through important details of batching and flattening to a single graph below.

A.3 Broadcast and pool operations

Consider how the message passing API could help us to find the total user spending on purchased items. As a preparation step, let's calculate for each "item" the latest price value and materialize the result to the graph tensor.

```
item_features = graph.node_sets["items"].get_features_dict()
item_features["latest_price"] = item_features["price"][:, :1].values
graph = graph.replace_features(node_sets={"items": item_features})
print(graph.node_sets["items"])
```

User spending can now be calculated by first computing purchase prices by broadcasting item latest prices to the "purchase" edges. The total user spendings are computed by sum-pooling purchase prices to their users.

```
purchase_prices = tfgnn.broadcast_node_to_edges(
    graph, "purchased", tfgnn.SOURCE, feature_name="latest_price")
total_user_spendings = tfgnn.pool_edges_to_node(
    graph,
    "purchased",
    tfgnn.TARGET,
    reduce_type="sum",
    feature_value=purchase_prices)
print(total_user_spendings)
```

Note that API allows to reference feature values either by their name and by their values. For latter it is not required that feature exists in the graph tensor as soon as its shape is correct.

Message passing is also supported between the graph context and any node set or edge set. As an example, let's compare individual user spendings to the maximum amount spent by any users. The code below first max-pool all individual user spendings and then broadcast them back to "users" to compute fractions.

```
max_amount_spend = tfgnn.pool_nodes_to_context(
    graph, "users", feature_value=total_user_spendings,
    reduce_type="max")
max_amount_spend = tfgnn.broadcast_context_to_nodes(
    graph, "users", feature_value=max_amount_spend)
print(total_user_spendings / max_amount_spend)
```

A.4 Implementing GATv2

The following code snippet shows the essence of how GATv2 [7] has been implemented in TF-GNN. It performs attention from many senders to one receiver.

In the simplest case, the convolution is applied to an edge set whose target nodes are the receivers, and each receiver attends to the adjacent source nodes. However, TF-GNN lets you configure this in many ways: (1) The roles of source and target nodes can be swapped by setting the `receiver_tag`; for example, consider how an edge set of hyperlinks between HTML docs can reasonably be used in either direction. (2) The sender value can be supplied by the neighbor node, the connecting edge, or both; this is controlled by the `sender_*_feature` args, at least one of which must not be `None`. (3) Attention can also happen with `receiver_tag=tfgnn.CONTEXT` and either all nodes or all edges

in the respective graph component as senders. This comes in handy for attention in the context update of a Graph Network [5] or for readout of a graph-level feature by smart pooling of node or edge states.

All those case distinctions are handled by the superclass, which passes the suitable broadcast and pool ops as arguments into the actual `GATv2Conv.convolve()` method.

```
class GATv2Conv(tfgnn.keras.layers.AnyToAnyConvolutionBase):

    def __init__(self, *,
                 num_heads, per_head_channels,
                 receiver_tag=None, # SOURCE, TARGET or CONTEXT.
                 receiver_feature=tfgnn.HIDDEN_STATE, # Required.
                 sender_node_feature=tfgnn.HIDDEN_STATE, # Set None to disable.
                 sender_edge_feature=None, # Set tfgnn.HIDDEN_STATE to enable.
                 attention_activation="leaky_relu", activation="relu",
                 edge_dropout=0., **kwargs):
        # Save initializer args.
        super().__init__(
            receiver_tag=receiver_tag, receiver_feature=receiver_feature,
            sender_node_feature=sender_node_feature,
            sender_edge_feature=sender_edge_feature,
            extra_receiver_ops={"softmax": tfgnn.softmax}, **kwargs)
        self._num_heads = num_heads
        self._per_head_channels = per_head_channels
        self._edge_dropout_layer = tf.keras.layers.Dropout(edge_dropout)
        self._attention_activation = tf.keras.activations.get(attention_activation)
        self._activation = tf.keras.activations.get(activation)
        # Create the transformations for the query input in all heads.
        self._w_query = tf.keras.layers.Dense(per_head_channels * num_heads)
        # Create the transformations for value input from sender nodes and edges.
        self._w_sender_node = self._w_sender_edge = None
        if self.takes_sender_node_input:
            self._w_sender_node = tf.keras.layers.Dense(per_head_channels * num_heads)
        if self.takes_sender_edge_input:
            self._w_sender_edge = tf.keras.layers.Dense(
                per_head_channels * num_heads,
                use_bias=self._w_sender_node is None) # Avoid two biases.
        # Create the transformation to attention scores.
        self._attention_logits_fn = tf.keras.layers.experimental.EinsumDense(
            "...ik,ki->...i", output_shape=(num_heads,))

    def convolve(self, *,
                 sender_node_input, sender_edge_input, receiver_input,
                 broadcast_from_sender_node, broadcast_from_receiver,
                 pool_to_receiver, extra_receiver_ops, training):
        # Form the attention query for each head.
        query = broadcast_from_receiver(self._split_heads(self._w_query(
            receiver_input)))
        # Form the attention value for each head.
        value_terms = []
        if sender_node_input is not None:
            value_terms.append(broadcast_from_sender_node(
                self._split_heads(self._w_sender_node(sender_node_input))))
        if sender_edge_input is not None:
            value_terms.append(
                self._split_heads(self._w_sender_edge(sender_edge_input)))
        value = tf.add_n(value_terms)
        # Compute the attention coefficients.
        attention_features = self._attention_activation(query + value)
        logits = tf.expand_dims(self._attention_logits_fn(attention_features), -1)
        attention_coefficients = extra_receiver_ops["softmax"](logits)
        attention_coefficients = self._edge_dropout_layer(attention_coefficients)
        # Apply the attention coefficients to the transformed query.
        messages = value * attention_coefficients
        pooled_messages = pool_to_receiver(messages, reduce_type="sum")
        # Apply the nonlinearity.
        return self._activation(self._merge_heads(pooled_messages))

    # The following helpers map forth and back between tensors with...
    # - a separate heads dimension: shape [..., num_heads, channels_per_head],
    # - all heads concatenated: shape [..., num_heads * channels_per_head].
    def _split_heads(self, tensor):
        extra_dims = tensor.shape[1:-1] # Possibly empty.
        new_shape = (-1, *extra_dims, self._num_heads, self._per_head_channels)
        return tf.reshape(tensor, new_shape)

    def _merge_heads(self, tensor):
        extra_dims = tensor.shape[1:-2] # Possibly empty.
        merged_dims = tensor.shape[-2:]
```

```
new_shape = (-1, *extra_dims, merged_dims.num_elements())  
return tf.reshape(tensor, new_shape)
```