Ideas from GNN

Tianyue Li

June 2022

1 Starting point

Recently, I was reading some paper on the topic of Graph Neural Network (GNN) and I found that it is very likely to apply the GNN knowledge into our provenance tracking method. From the beginning of this project, I was always trying to solve this problem from a graph view and I found that GNN is really an exciting idea to include what we have thought so far and what can lead us to next stage. **Note that it does not necessarily mean that we will adopt a ML model as our central idea, we are now just trying to adapt some graph knowledge into our domain**

2 Definition

2.1 Definition of graph

To get start with, we need to first define what vertex,edge in a graph represent in the tensor domain. And what information should they carry in that case.

We denote a graph as $G = (V, E)$, where $V = 1, 2, ..., N$, and N is the total number of vertexes, $E \subseteq V \times V$. **For a vertex, it should represent a tensor.** And the edge is a kind of relationship, which is not an operation. **An operation is a combination of edges and vertexes(a sub graph basically).**

2.2 Definition of triplet

What is the most important thing for both two methods is to understand the concept of triplet. For a knowledge graph, triplet is basically a tuple (h, r, t), in which h represent the head,r represent the relation, t represent the tail. And h and t both satisfied $h, t \subseteq V$, and r should satisfy $r \subseteq \mathcal{R}$. This $\mathcal{R}$ will be given more detail in the abstract semantic method.

This definition is both useful for ML method and abstract semantic method. (Not sure if we should call it that way).

3 Method

3.1 ML Method

This part is very easy to imagine, we just get millions of generated graph and feed it to the GNN model. Everything could follows a typical GNN construction procedures.

For the vertexes, we will make it an $N \times D$ matrix. N represents the number of vertex, and D is the feature tensor.(More detail please refer to GNN concepts). And for the relationship$\mathcal{R}$, it will use the same concept as what has been told in 2.2. The most importance thing here is how we define the task. And I would say we are finding the relationship R in the triplet (h, r, t), in which $h = e_{in}$ and $t = e_{out}$. Namely constructing invisible edges.

I will give more explanation on the Abstract Semantics method.

3.2 Abstract Semantics

3.2.1 Basic concepts

As mentioned before, the core of this idea is the construction of triplet. Specifically, the vertexes refer to tensors. **There is no difference between vertexes**. However, we will specify the **sub-operation relation, which is what edge refers to.** Before I give some examples. I will first give some definition for the sub-operation relation.

Definition 1: A sub-operation relation is a direct relation between two tensors, and is a single directed relation. Some of such relation involves $\mathcal{G}$(Give elements), $\mathcal{T}$(Transform elements), $\mathcal{N}$(Not give element).

Note that the definition of sub-operation relation is the element r within the triplet (h, r, t). Next, we define what $\mathcal{G}$, $\mathcal{T}$, $\mathcal{N}$ are separately.

Definition of $\mathcal{G}$: Between the process of transformation from $T_1 \to T_2$, in which $\to$ is composed of a single sub-operation. If $\exists\ t_i \in T_2$ such that there $\exists t_j \in T_1$, $t_i \cong t_j$ AND t_i comes directly from t_j. Here, $\cong$ means the values of two elements are the same, but the date type can be different.

Definition of $\mathcal{T}$: Between the process of transformation from $T_1 \to T_2$, in which $\to$ is composed of a single sub-operation. If $\exists\ t_i \in T_2$ and there $\exists t_j \in T_1$, such that $t_i = f(t_j, ...)$ AND $t_i \neq t_j$. Here, function f is composed of some simple arithmetic operation, which includes $+, -, *, /, log, exp, floor, matmul$. And ... inside f means that there might exist some other elements from the T_1 or other head which has the tail T_2 working as parameters.

Below is an example which contains two triplets (in1,$\mathcal{T}$,out1),(in2,$\mathcal{T}$,out1).

Example 1: tf.add

We color $\mathcal{T}$(Transform elements) to be green.

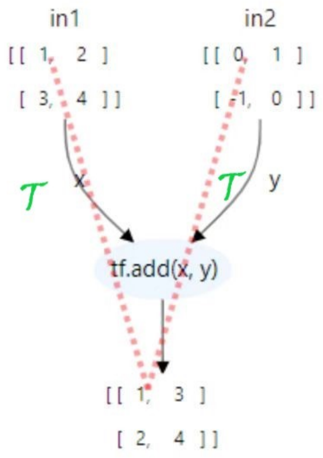

Figure 1: tf.add

Definition of $\mathcal{N}$: Between the process of transformation from $T_1 \rightarrow T_2$, in which $\rightarrow$ is composed of a single sub-operation. If $\forall\ t_i \in T_2$, there $!\exists t_j \in T_1$, such that t_i has its provenance to be t_j. Typical examples includes, tf.ones, tf.ones$_like, indexing tensor in tf.gather, tf.greater (type changes from interger to boolean)$.

Below is an typical and often mentioned example.

Example 2: tf.gather

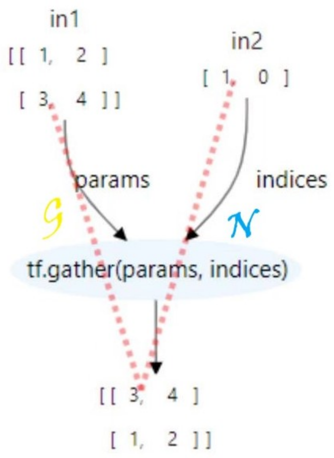

Figure 2: tf.gather

Here we give another example. Just to make things clear.
Example 3:tf.pad

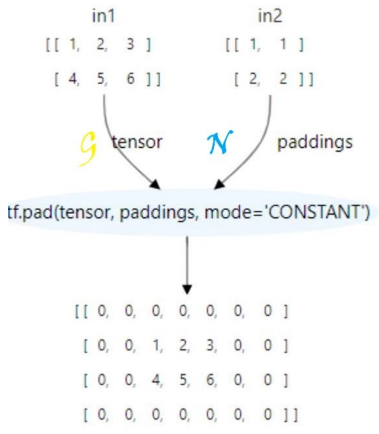

Figure 3: tf.pad

The reason I do not choose to classify vertexes (tensors) into different functions (like what Kellen does in his method, assigning each vertexes as $b|r|i$) is that **it is very common for a tensor function differently as a head and a tail**. Please see the example below:

Counter-example:

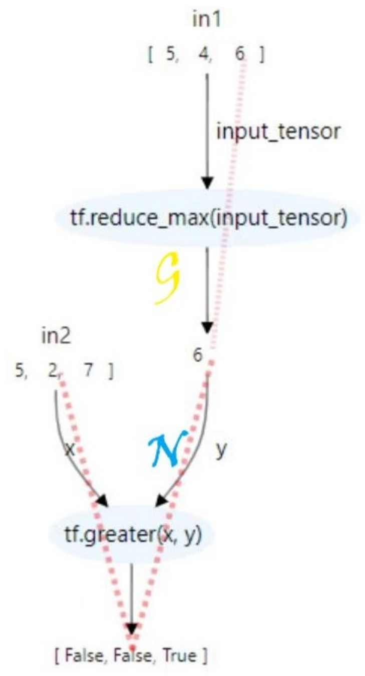

Figure 4: operation composition

Here, [6] both servers as an regular tensor and a boolean tensor in Kellen's example. But the transformation can be easily written as a combination of three triplets. (in1,$\mathcal{G}$,6), (6,$\mathcal{N}$,out), (in2,$\mathcal{N}$,out).

3.2.2 Explanation from User specification

The build of this method is heavily dependent on the style of how user will input. And let us specify the user input style as follows:

User Input Specification. User will specify a list of input tensors T_{in}, and an output tensor T_{out}. And user will also specify a **partial element-wise specification**. In the partial element-wise specification, the user will make it clear if an element in the output tensor is formed **directly by transferring from another tensor's elements | by transforming from other tensors elements | not coming from any element in the input tensor.**

Since it will be hard for user to input all the provenance output's elements in a very detailed style, I choose to let the user only **Partially input** the provenance. In addition, I further make the user only need to input a general provenance of elements, which contains only three classes $\mathcal{G}$, $\mathcal{T}$ and $\mathcal{N}$. In this way, it will make the user input style even more concise. And this is all what the program need to calculate the provenance.

3.2.3 Some abstract semantics derivation

In this section, I will provide some trivial abstract semantics based on the $\mathcal{G}$, $\mathcal{T}$, $\mathcal{N}$. Note that $\rightarrow$ means a single sub-operation.

$$\mathcal{G}- > \mathcal{T} := T$$
$$\mathcal{X}- > \mathcal{N} := N$$
$$\mathcal{N}- > \mathcal{X} := N$$
$$\mathcal{X}- > \mathcal{G} := X$$
$$\mathcal{T}- > \mathcal{T} := T$$

$\mathcal{X}$, means either $\mathcal{G}$ or $\mathcal{T}$ or $\mathcal{N}$.
Note that the above equations can be combined together:
For example,

$$\mathcal{G}- > \mathcal{T}- > \mathcal{T} := T$$

3.2.4 Over-approximation in this abstract semantics

By summarizing a sub-operation as $\mathcal{G}$, we actually assume that there is a chance for all elements in the input tensor to be passed. However, in some operation that function as $\mathcal{G}$, like tf.max(),tf.top-k(), only parts of the elements are passed down. Therefore, the over-approximation in this abstract semantics is that we assume all the elements in the input tensor are passed down within a $\mathcal{G}$ sub-operation.

3.2.5 Statistics of Tensorflow operations

I have tried to classify all sub-operations of **104** operations mentioned in the TF-coder paper into $\mathcal{G}$ or $\mathcal{T}$ or $\mathcal{N}$. And the result is attached below. Also, the picture of raw data is attached in the appendix.

Reference name	$\mathcal{G}$	$\mathcal{T}$	$\mathcal{N}$
Sub-operation numbers	44	45	58
operation involves	39	36	48

Table 1: Statistics of sub-operation

The first row represents the sub-operation that involves $\mathcal{G}$ or $\mathcal{T}$ or $\mathcal{N}$. The second row represents all the operation that contains sub-operation as $\mathcal{G}$ or $\mathcal{T}$ or $\mathcal{N}$. It can be seen that $\mathcal{N}$ contains more than all the two others. Therefore, **we may further divide this class to make the classification more precise and easy to trace for the program**.

3.2.6 Possible improvement on the abstract semantics

1. we can divide the $\mathcal{N}$ class to make the classification more precise.
2. We can use the sub-operation classification to divide the operations. For example, tf.add, tf.matmul can be classified as the same class since they both use two $\mathcal{T}$ class sub-operations. This can do much help in searching.

4 Appendix A: the operations

A SUPPORTED OPERATIONS IN TF-CODER

Below is the list of 134 operations currently supported by TF-Coder. We did not cherrypick the operations to support; in fact, out of the 134 supported operations, only 59 are used in TF-Coder's solutions to our benchmark tasks.

```
General TensorFlow functions:
-----------------------------
tf.abs(x)
tf.add(x, y)
tf.add_n(inputs)
tf.argmax(input, axis)
tf.argmin(input, axis)
tf.argsort(values, axis, stable=True)
tf.argsort(values, axis, direction='DESCENDING', stable=True)
tf.boolean_mask(tensor, mask)
tf.broadcast_to(input, shape)
tf.cast(x, dtype)
tf.clip_by_value(t, clip_value_min, clip_value_max)
tf.concat(values, axis)
tf.constant(value)
tf.constant(value, dtype)
tf.divide(x, y)
tf.equal(x, y)
tf.exp(x)
tf.expand_dims(input, axis)
tf.eye(num_rows)
tf.eye(num_rows, num_columns)
tf.eye(num_rows, dtype)
tf.fill(dims, value)
tf.gather(params, indices)
tf.gather(params, indices, axis, batch_dims)
tf.gather_nd(params, indices)
tf.gather_nd(params, indices, batch_dims)
tf.greater(x, y)
tf.greater_equal(x, y)
tf.math.bincount(arr)
tf.math.ceil(x)
tf.math.count_nonzero(input)
tf.math.count_nonzero(input, axis)
tf.math.cumsum(x, axis)
tf.math.cumsum(x, axis, exclusive=True)
tf.math.divide_no_nan(x, y)
tf.math.floor(x)
tf.math.log(x)
tf.math.negative(x)
tf.math.reciprocal(x)
tf.math.reciprocal_no_nan(x)
tf.math.segment_max(data, segment_ids)
tf.math.segment_mean(data, segment_ids)
tf.math.segment_min(data, segment_ids)
tf.math.segment_prod(data, segment_ids)
tf.math.segment_sum(data, segment_ids)
tf.math.squared_difference(x, y)
tf.math.top_k(input, k)
tf.math.unsorted_segment_max(data, segment_ids, num_segments)
tf.math.unsorted_segment_mean(data, segment_ids, num_segments)
tf.math.unsorted_segment_min(data, segment_ids, num_segments)
tf.math.unsorted_segment_prod(data, segment_ids, num_segments)
tf.math.unsorted_segment_sum(data, segment_ids, num_segments)
tf.matmul(a, b)
tf.maximum(x, y)
tf.minimum(x, y)
tf.multiply(x, y)
tf.not_equal(x, y)
tf.one_hot(indices, depth)
tf.ones(shape)
tf.ones_like(input)
tf.pad(tensor, paddings, mode='CONSTANT')
tf.pad(tensor, paddings, mode='CONSTANT', constant_values)
tf.pad(tensor, paddings, mode='REFLECT')
tf.pad(tensor, paddings, mode='SYMMETRIC')
tf.range(start)
tf.range(start, limit, delta)
tf.reduce_any(input_tensor, axis)
tf.reduce_max(input_tensor)
tf.reduce_max(input_tensor, axis)
tf.reduce_mean(input_tensor)
tf.reduce_mean(input_tensor, axis)
tf.reduce_min(input_tensor)
tf.reduce_min(input_tensor, axis)
tf.reduce_prod(input_tensor, axis)
tf.reduce_sum(input_tensor)
tf.reduce_sum(input_tensor, axis)
tf.reshape(tensor, shape)
tf.reverse(tensor, axis)
tf.roll(input, shift, axis)
tf.round(x)
tf.searchsorted(sorted_sequence, values, side='left')
tf.searchsorted(sorted_sequence, values, side='right')
tf.sequence_mask(lengths)
tf.sequence_mask(lengths, maxlen)
tf.shape(input)
tf.sign(x)
tf.sort(values, axis)
tf.sort(values, axis, direction='DESCENDING')
tf.sqrt(x)
tf.square(x)
tf.squeeze(input)
tf.squeeze(input, axis)
tf.stack(values, axis)
tf.subtract(x, y)
tf.tensordot(a, b, axes)
tf.tile(input, multiples)
tf.transpose(a)
tf.transpose(a, perm)
tf.unique_with_counts(x)
tf.unstack(value, axis)
tf.where(condition)
tf.where(condition, x, y)
tf.zeros(shape)
tf.zeros_like(input)
```