Covariant Compositional Networks and GraphFlow Deep Learning Framework

Hy Truong Son, Nguyen Duc Hai

Department of Computer Science, The University of Chicago

hytruongson@uchicago.edu, ndhai@uchicago.edu

Abstract

In this paper, we propose Covariant Compositional Networks (CCNs), the state-of-the-art generalized convolution graph neural network for learning graphs. By applying higher-order representations and tensor contraction operations that are permutation-invariant with respect to the set of vertices, CCNs address the representation limitation of all existing neural networks for learning graphs in which permutation invariance is only obtained by summation of feature vectors coming from the neighbors for each vertex via well-known message passing scheme. To efficiently implement graph neural networks and high-complexity tensor operations in practice, we designed our custom Deep Learning framework in C++ named GraphFlow that supports dynamic computation graphs, automatic and symbolic differentiation as well as tensor/matrix implementation in CUDA to speed-up computation with GPUs. For an application of graph neural networks in quantum chemistry and molecular dynamics, we investigate the efficiency of CCNs in estimating Density Functional Theory (DFT) that is the most successful and widely used approach to compute the electronic structure of matter but significantly expensive in computation. We obtain a very promising result and outperform other state-of-the-art models in Harvard Clean Energy Project and QM9 molecular dataset.

Index Terms: graph neural network, message passing, representation theory, density functional theory, molecular dynamics

1. Introduction

In the field of Machine Learning, standard objects such as vectors, matrices, tensors were carefully studied and successfully applied into various areas including Computer Vision, Natural Language Processing, Speech Recognition, etc. However, none of these standard objects are efficient in capturing the structures of molecules, social networks or the World Wide Web which are not fixed in size. This arises the need of graph representation and extensions of Support Vector Machine and Convolution Neural Network to graphs.

To represent graphs in general and molecules specifically, the proposed models must be *permutation-invariant* and *rotation-invariant*. In addition, to apply kernel methods on graphs, the proposed kernels must be *positive semi-definite*. Many graph kernels and graph similarity functions have been introduced by researchers. Among them, one of the most successful and efficient is the Weisfeiler-Lehman graph kernel which aims to build a multi-level, hierarchical representation of a graph [1]. However, a limitation of kernel methods (see section 2.1) is quadratic space usage and quadratic time-complexity in the number of examples. In this paper, we address this drawback by introducing Weisfeiler-Lehman graph features in combination with Morgan circular fingerprints in sections 2.2 and 2.3. The common idea of family of Weisfeiler-Lehman graph kernel is hashing the sub-structures of a graph. Extending this idea, we come to the simplest form of graph neural networks in which the *fixed* hashing function is replaced by a *learnable* one as a non-linearity mapping. (See section 2.4.) We detail the graph neural network baselines such as Neural Graph Fingerprint [6] and Learning Convolutional Neural Networks [9] in sections 2.4.1 and 2.4.2. In the context of graphs, the sub-structures can be considered as a set of vertex feature vectors. We ultilize the convolution operation by introducing higher-order representations for each vertex, from zeroth-order as a vector to the first-order as a matrix and the second-order as a 3rd order tensor in section 3.1. Also in this section, we introduce the notions of tensor contractions (see section 3.5) and tensor products (see section 3.3) to keep the orders of tensors manageable without exponentially growing. Our generalized convolution graph neural network is named as Covariant Compositional Networks (CCNs). We propose 2 specific algorithms that are first-order CCN in section 3.6 and second-order CCN in section 3.7. It is trivial that high-order tensors cannot be stored explicitly in memory of any conventional computers. To make tensor computations feasible, we build *Virtual Indexing System* that returns value of each tensor element given the corresponding index, and allows efficient GPU implementation. (See section 3.4.).

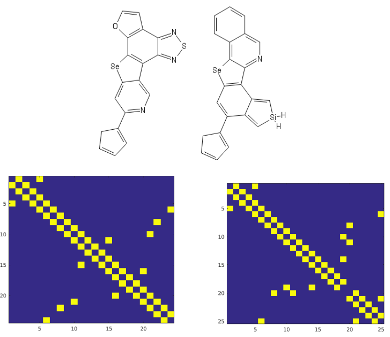
Current Deep Learning frameworks including TensorFlow [12], PyTorch [13], Mxnet [15], Theano [14], etc. showed their limitations for constructing dynamic computation graphs along with specialized tensor operations. It leads to the need of a flexible programming framework for graph neural networks addressing both these drawbacks. With this motivation, we designed our Deep Learning framework in C++ named GraphFlow for our long-term Machine Learning research. All of our experiments have been implemented efficiently within GraphFlow. In addition, GraphFlow is currently being parallelized with CPU/GPU multi-threading. Implementation of GraphFlow is mentioned in section 4. Finally, we apply our methods to the Harvard Clean Energy Project (HCEP) [2] and QM9 [17] molecular dataset. The visualizations, experiments and empirical results are detailed in section 5. Section 6 is our conclusion and future research direction.

2. Background work

2.1. Definition of graph kernel

Kernel-based algorithms, such as Gaussian processes [22], support vector machines [23], and kernel PCA [24], have been widely used in the statistical learning community. Given the input domain $\mathcal{X}$ that is some nonempty set, the common idea is to express the correlations or the similarities between pairs of

Figure 1: *Two molecules with adjacency matrices in HCEP*



$C_{18}H_9N_3OSSe$ $C_{22}H_{15}NSeSi$

points in $\mathcal{X}$ in terms of a kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. The kernel function $k(\cdot,\cdot)$ is required to satisfy that for all $x, x' \in \mathcal{X}$,

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle \quad (1)$$

where $\Phi : \mathcal{X} \to \mathcal{H}$ maps from the input domain $\mathcal{X}$ into some dot product space $\mathcal{H}$. We call Φ as a feature map and $\mathcal{H}$ as a feature space. Given a kernel k and inputs $x_1, .., x_n \in \mathcal{X}$, the $n \times n$ matrix

$$K \triangleq (k(x_i, x_j))_{ij} \quad (2)$$

is called the Gram matrix (or kernel matrix) of kernel function k with respect to $x_1, .., x_n$. A symmetric matrix $K \in \mathbb{R}^{n \times n}$ satisfying

$$c^T K c = \sum_{i,j} c_i c_j K_{ij} \geq 0 \quad (3)$$

for all $c \in \mathbb{R}^n$ is called positive definite. If equality in 3 happens when $c_1 = .. = c_n = 0$, then K is called strictly positive definite. A symmetric function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is called a positive definite kernel on $\mathcal{X}$ if

$$\sum_{i,j} c_i c_j k(x_i, x_j) \geq 0 \quad (4)$$

holds for any $n \in \mathbb{N}$, $\{x_1, .., x_n\} \subseteq \mathcal{X}^n$ and $c \in \mathbb{R}^n$. The inequality 4 is equivalent with saying the Gram matrix K of kernel function k with respect to inputs $x_1, .., x_n \in \mathcal{X}$ is positive definite.

A graph kernel $\mathcal{K}_{graph} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a positive definite kernel having the input domain $\mathcal{X}$ as a set of graphs. Given two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Assume that each vertex is associated with a feature vector $f : V \to \Omega$ where Ω is a vector space. A positive definite graph kernel $\mathcal{K}_{graph}$ between G_1 and G_2 can be written as:

$$\mathcal{K}_{graph} \triangleq \frac{1}{|V_1|} \cdot \frac{1}{|V_2|} \cdot \sum_{v_1 \in V_1} \sum_{v_2 \in V_2} k_{base}(f(v_1), f(v_2)) \quad (5)$$

where k_{base} is any base kernel defined on vector space Ω, and can be:

- Linear: $k_{base}(x, y) \triangleq \langle x, y \rangle_{norm} \triangleq x^T y / (\|x\| \cdot \|y\|)$
- Quadratic: $k_{base}(x, y) \triangleq (\langle x, y \rangle_{norm} + q)^2$ where $q \in \mathbb{R}$
- Radial Basis Function (RBF): $k_{base}(x, y) \triangleq \exp(-\gamma \|x - y\|^2)$ where $\gamma \in \mathbb{R}$

2.2. Weisfeiler-Lehman graph isomorphism test [21]

Given two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ where V_1 and V_2 are the sets of vertices, E_1 and E_2 are the sets of edges. Suppose we have a mapping $l : V_1 \cup V_2 \to \Sigma$ that is a vertex labeling function giving label $l(v) \in \Sigma$ from the set of all possible labels Σ for each vertex $v \in V_1 \cup V_2$. Assuming that $|V_1| = |V_2|$ and $|E_1| = |E_2|$, the graph isomorphism test is defined as: Determine whether there exists a permutation on the vertex indices such that two graphs G_1 and G_2 are identical. Formally saying, we have to find a bijection between the set of vertices of G_1 and G_2, $\sigma : V_1 \to V_2$, such that

$$\forall (u, v) \in E_1 : (\sigma(u), \sigma(v)) \in E_2 \quad (6)$$

In addition, we can add one more contraint on the vertex labels such that

$$\forall v \in V_1 : l(v) = l(\sigma(v)) \quad (7)$$

The algorithm of Weisfeiler-Lehman (WL) graph isomorphism test is described as follows. We can see that if G_1 and G_2 are

Algorithm 1: Weisfeiler-Lehman iterations

```
Data: Given an undirected graph $G = (V, E)$, vertex
      labels $l(v) \in \Sigma$ for all $v \in V$, and $T \in \mathbb{N}$ as the
      number of Weisfeiler-Lehman iterations.
      Assuming that we have a perfect hashing function
      $h : \Sigma^* \to \Sigma$.
Result: Return $f_i(v) \in \Sigma^*$ for all $v \in V$ and $i \in [0, T]$.
1  for $i = 0 \to T$ do
2      Compute the multiset of labels $M_i(v)$, string $s_i(v)$
         and compressed label $f_i(v)$
3      for $v \in V$ do
4          if $i = 0$ then
5              $M_i(v) \leftarrow \varnothing$
6              $s_i(v) \leftarrow l(v)$
7              $f_i(v) \leftarrow h(s_i(v))$
8          else
9              $M_i(v) \leftarrow \{f_{i-1}(u) | u \in \mathcal{N}(v)\}$ where
                $\mathcal{N}(v) = \{u | (u, v) \in E\}$
10             Sort $M_i(v)$ in ascending order and
                 concatenate all labels of $M_i(v)$ into string
                 $s_i(v)$
11             $s_i(v) \leftarrow s_i(v) \oplus f_{i-1}(v)$ where $\oplus$ is
                 concatenation operation.
12             $f_i(v) \leftarrow h(s_i(v))$
13         end
14     end
15 end
```

isomorphic then the WL test always returns true. In the case G_1 and G_2 are not isomorphic, the WL test returns true with a small probability. In particular, the WL algorithm has been shown to be a valid isomorphism test for almost all graphs. Suppose that we have an efficient sorting algorithm $O(N \log_2 N)$ for a sequence of N items, and the time complexities for concatenation operations and computing the hashing functions are negligible. In algorithm 1, for each iteration i-th, each edge (u, v) is considered twice and $|\mathcal{N}(v)| \leq |V|$. Thus the time complexity of algorithm 1 is $O(T(|V| + |E| \log_2 |V|))$. In algorithm 2, $|F_i^{G_1}| = |V_1|$, the time complexity to sort $F_i^{G_1}$ is $O(|V_1| \log_2 |V_1|)$, and similarly for G_2. Therfore, the total time comlexity of WL isomorphism test is $O(T(|V| + |E|) \log_2 |V|)$ where $|V| = \max\{|V_1|, |V_2|\}$ and $|E| = |E_1| + |E_2|$.

Algorithm 2: Weisfeiler-Lehman graph isomorphism test

```
    Data: Given two undirected graphs $G_1 = (V_1, E_1)$ and
          $G_2 = (V_2, E_2)$ with vertex labels
          $l : V_1 \cup V_2 \to \Sigma$.
    Result: Return whether $G_1$ and $G_2$ are isomorphic.
 1  Apply algorithm 1 on graph $G_1$ to get $\{f_i^{G_1}(v)\}$
 2  Apply algorithm 1 on graph $G_2$ to get $\{f_i^{G_2}(v)\}$
 3  for $i = 0 \to T$ do
 4  |   $F_i^{G_1} \leftarrow \{f_i^{G_1}(v)|v \in V_1\}$
 5  |   $F_i^{G_2} \leftarrow \{f_i^{G_2}(v)|v \in V_2\}$
 6  |   Sort $F_i^{G_1}$ in ascending order
 7  |   Sort $F_i^{G_2}$ in ascending order
 8  |   if $F_i^{G_1} \not\equiv F_i^{G_2}$ then
 9  |   |   return $G_1$ and $G_2$ are not isomorphic
10  |   end
11  end
12  return $G_1$ are $G_2$ are isomorphic
```

2.3. Weisfeiler-Lehman graph kernel [1]

Based on algorithms 1 2 and equation 5, we introduce the following algorithm to compute the Weisfeiler-Lehman kernel between the two input graphs G_1 and G_2. In this case, G_1 and G_2 can have different numbers of vertices. The remaining question is: what would be the possible choices of vertex labels $l(v)$? One way to define the vertex labels is using the vertex degrees:

$$l(v) \triangleq |\{u|(u, v) \in E\}| = |\mathcal{N}(v)| \tag{8}$$

Suppose that the time complexity to compute the base kernel value between $f^{G_1}(v_1)$ and $f^{G_2}(v_2)$ is $O(T)$ for every pair of vertices (v_1, v_2). Thus the time complexity to compute the WL kernel value is $O(T|V|^2)$ where $|V| = \max\{|V_1|, |V_2|\}$. Therefore, the total time complexity of WL graph kernel algorithm is $O(T(|V|^2 + |E| \log_2 |V|))$ where $|E| = |E_1| + |E_2|$.

Algorithm 3: Weisfeiler-Lehman graph kernel

```
    Data: Given two undirected graphs $G_1 = (V_1, E_1)$ and
          $G_2 = (V_2, E_2)$ with vertex labels
          $l : V_1 \cup V_2 \to \Sigma$.
    Result: Return the WL kernel value.
 1  Apply algorithm 1 on graph $G_1$ to get $\{f_i^{G_1}(v)\}$
 2  Apply algorithm 1 on graph $G_2$ to get $\{f_i^{G_2}(v)\}$
 3  for $v \in V_1$ do
 4  |   $f^{G_1}(v) \leftarrow \bigoplus_{i=0}^{T} f_i^{G_1}(v)$
 5  end
 6  for $v \in V_2$ do
 7  |   $f^{G_2}(v) \leftarrow \bigoplus_{i=0}^{T} f_i^{G_2}(v)$
 8  end
 9  $\mathcal{K}_{graph}(G_1, G_2) \leftarrow$
      $\frac{1}{|V_1|} \cdot \frac{1}{|V_2|} \cdot \sum_{v_1 \in V_1} \sum_{v_2 \in V_2} k_{base}(f^{G_1}(v_1), f^{G_2}(v_2))$
10  return $\mathcal{K}_{graph}(G_1, G_2)$
```

2.4. Graph Neural Networks

In this section, we discuss about the two state-of-the-art graph neural networks that are our baseline models: Neural Graph Fingerprint (NGF) proposed by Duvenaud and colleagues in their NIPS 2015 paper [6], and Learning Convolutional Neural Networks (LCNN) proposed by Niepert and colleagues in their ICML 2016 paper [9]. However, these two models have their own drawbacks:

- NGF has its limitation in representation power since each vertex is only represented by a multi-channel vector, that means each channel is represented by only a single scalar. We classify this type of representation as zeroth-order. To empower the vertex representation, we propose the first-order and second-order representations in which each channel is representated by a vector and a matrix, consequentially the vertex representations are a matrix and 3rd order tensor, respectively.
- LCNN has its limitation in permutation invariance because it uses Weisfeiler-Lehman algorithm to rank the set of vertices into a particular ordering. However, finding an optimal ordering of the set of vertices is an NP-hard problem. That means LCNN is not invariant to the permuation of vertices. To address this issue, we apply tensor contraction and tensor product operations over high-order representations of vertices such that these operations are perfectly equivariant.

2.4.1. *Neural Graph Fingerprint [6]*

First of all, we define a simple form of message passing scheme. Given an input graph $G = (V, E, A)$, where V is the set of vertices, E is the set of edges and matrix $A \in \{0, 1\}^{|V|\times|V|}$ is the corresponding adjacency matrix. The goal is to learn an unknown class of functions parameterized by $\{W_1, .., W_T, u\}$ in the following scheme:

1. The inputs are vectors $f(v) \in \mathbb{R}^d$ for each vertex $v \in V$. We call the vector embedding f the multi-dimensional vertex label function.
2. We assume some learnable weight matrix $W_i \in \mathbb{R}^{d\times d}$ associating with level i-th of the neural network. For T levels, we update the vector stored at vertex v using W_i.
3. Finally, we assume some learnable weight vector $u \in \mathbb{R}^d$. We add up the iterated vertex labels and dot product the result with u. This can be considered as a linear regression on top of the graph neural network.

More formally, we define the T-iteration label propagation algorithm on graph G. Let $h_t(v) \in \mathbb{R}^d$ be the vertex embedding of vertex v at iteration $t \in \{0, T\}$. At $t = 0$, we initialize $h_0(v) = f(v)$. At $t \in \{1, .., T\}$, we update h_{t-1} to h_t at a vertex v using the values on v's neighbors:

$$h_t(v) = h_{t-1}(v) + \frac{1}{|\mathcal{N}(v)|} \sum_{w \in \mathcal{N}(v)} h_{t-1}(w) \tag{9}$$

where $\mathcal{N}(v) = \{w \in V|(v, w) \in E\}$ denotes the set of adjacent vertices to v. We can write the label propagation algorithm in a matrix form. Let $H_t \in \mathbb{R}^{|V|\times d}$ denote the vertex embedding matrix in which the v-th row of H_t is the embedding of vertex v at iteration t. Equation 9 is equivalent to:

$$H_t = (I_{|V|} + D^{-1} \cdot A) \cdot H_{t-1} \tag{10}$$

where $I_{|V|}$ is the identity matrix of size $|V| \times |V|$ and D is the diagonal matrix with entries equal to the vertex degrees. Note that's is also common to define another label propagation algorithm via the normalized graph Laplacian [7]:

$$H_t = (I_{|V|} - D^{-1/2}AD^{-1/2}) \cdot H_{t-1} \qquad (11)$$

From the label propagation algorithms, we build the simplest form of graph neural networks [5, 6, 7]. Suppose that iteration t is associated with a learnable matrix $W_t \in \mathbb{R}^{d \times d}$ and a component-wise nonlinearity function σ; in our case σ is the sigmoid function. We imagine that each iteration now becomes a layer of the graph neural network. We assume that each graph G has input labels f and a learning target $\mathcal{L}_G \in \mathbb{R}$. The forward pass of the graph neural network (GNN) is described by algorithm 4. Learnable matrices W_i and learnable vector u are optimized by the Back-Propagation algorithm as done when training a conventional multi-layer feed-forward neural network.

Algorithm 4: Forward pass of GNN

```
Data: Given an undirected graph $G = (V, E, A)$ where
      $V$, $E$ and $A$ are the set of vertices, the set of
      edges and the adjacency matrix, respectively. The
      number of layers is $T \in \mathbb{N}$.
Result: Construct the corresponding neural network.
1 Initialize $W_0, W_1, .., W_T \in \mathbb{R}^{d \times d}$
2 Layer 0: $L_0 = \sigma(H_0 \cdot W_0)$
3 Layer $t \in \{1, .., T\}$: $L_t = \sigma(H_t \cdot W_t)$
4 Compute the graph feature: $f_G = \sum_{v \in V} L_T(v) \in \mathbb{R}^d$
5 Linear regression on layer $T + 1$
6 Minimize: $\|\langle u, f_G \rangle - \mathcal{L}_G\|_2^2$ where $u \in \mathbb{R}^d$ is learnable
```

To empower Neural Graph Fingerprint, we can also introduce quadratic and cubic aggregation rules that can be considered a special simplified form of tensor contractions. In detail, the linear aggregation rule can be defined as summation of feature vectors in a neighborhood $\mathcal{N}(v)$ of vertex v at level $l - 1$ to get a permutation invariant representation of vertex v at level l:

$$\phi_l^{linear}(v) = \sum_{w \in \mathcal{N}(v)} h_{l-1}(w)$$

where $\phi_l^{linear}(v) \in \mathbb{R}^d$ and $h_{l-1}(w) \in \mathbb{R}^d$ are still in zeroth order representation such that each channel of d channels is represented by a single scalar. Extending this we get the quadratic aggregation rule for $\phi_l^{quadratic}(v)$:

$$\phi_l^{quadratic}(v) = diag\left(\sum_{u \in \mathcal{N}(v)} \sum_{w \in \mathcal{N}(v)} h_{l-1}(u) h_{l-1}(w)^T \right)$$

where $h_{l-1}(u)h_{l-1}(w)^T \in \mathbb{R}^{d \times d}$ is the outter-product of level $(l-1)$-th representation of vertex u and w in the neighborhood $\mathcal{N}(v)$. Again $\phi_l^{quadratic}(v) \in \mathbb{R}^d$ is still in zeroth-order. Finally, we extend to the cubic aggregation rule for $\phi_l^{cubic}(v)$:

$$\phi_l^{cubic}(v) = diag\left(\sum_{u,w,t \in \mathcal{N}(v)} h_{l-1}(u) \otimes h_{l-1}(w) \otimes h_{l-1}(t) \right)$$

where $h_{l-1}(u) \otimes h_{l-1}(w) \otimes h_{l-1}(t) \in \mathbb{R}^{d \times d \times d}$ is the tensor product of 3 rank-1 vectors, and we obtain zeroth-order $\phi_l^{cubic}(v) \in \mathbb{R}^d$ by taking the diagonal of the 3rd order result tensor.

Moreover, it is not a natural idea to limit the neighborhood $\mathcal{N}(v)$ to only the set of adjacent vertices of v. Another way to extend $\mathcal{N}(v)$ is to use different neighborhoods at different levels / layers of the network, for example:

- At level $l = 0$: $\mathcal{N}_0(v) = \{v\}$
- At level $l > 0$:

  $$\mathcal{N}_l(v) = \mathcal{N}_{l-1}(v) \cup \bigcup_{w \in B(v,1)} \mathcal{N}_{l-1}(w)$$

  where $B(v, 1)$ denotes the set of vertices are at the distance 1 from the center v.

In section 3.1, we will discuss further this hierarchical extension of neighborhood and definition of a receptive field in Covariant Compositional Networks.

2.4.2. Learning Convolutional Neural Networks [9]

The idea of LCNN can be summarized as *flattening* a graph into a fixed-size sequence. Suppose that the maximum number of vertices over the whole dataset is N. Consider an input graph $G = (V, E)$. If $|V| < N$ then we add $N - |V|$ *dummy vertices* into V such that every graph in the dataset has the same number of vertices. For each vertex $v \in V$, LCNN fixes the size of its neighborhood $\Omega(v)$ as K. In the case $|\Omega(v)| < K$, again we add $K - |\Omega(v)|$ dummy vertices into $\Omega(v)$ to ensure that every neighborhood of every vertex has exactly the same number of vertices. Let $d : V \times V \to \{0, .., |V| - 1\}$ denote the shortest-path distance between any pair of vertices in G. Let $\sigma : V \to \Re$ denote the sub-optimal hashing function obtained from Weisfeiler-Lehman graph isomorphism test. Based on σ, we can obtain a sub-optimal ranking of vertices. The neighborhood $\Omega(v)$ of vertex v is constructed by algorithm 5. We also have algorithm 6 to flatten the input graph G as follows into a sequence of $N \times K$ vertices. Suppose that each vertex is associated with a fixed-size input feature vector of L channels. By the graph flattening algorithm 6, we can produce a feature matrix of size $L \times (NK)$. We can apply the standard convolutional

Algorithm 5: Construct Neighborhood of $v \in V$

```
Data: Given an undirected graph $G = (V, E, A)$ and a
      vertex $v \in V$.
Result: Construct the receptive field $\Omega(v)$.
1  $\Omega(v) \leftarrow \emptyset$
2  for $l \in 0, .., |V| - 1$ do
3      for $w \in V$ do
4          if $d(v, w) = l$ then
5              $\Omega(v) \leftarrow \Omega(v) \cup \{w\}$
6          end
7      end
8      if $|\Omega(v)| \geq K$ then
9          break
10     end
11 end
12 if $|\Omega(v)| < K$ then
13     Add $K - |\Omega(v)|$ dummy vertices into $\Omega(v)$
14 end
15 Suppose $\Omega(v) = \{v_1, .., v_K\}$
16 Sort $\Omega(v) \leftarrow \{v_{i_1}, .., v_{i_K}\}$ such that $\sigma(v_{i_t}) < \sigma(v_{i_{t+1}})$
17 return $\Omega(v)$
```

Algorithm 6: Flattening the graph

```
Data: Given an undirected graph $G = (V, E)$.
Result: Sequence $S$ of $N \times K$ vertices.
1 Suppose that $V = \{v_1, .., v_{|V|}\}$
2 Sort $\overline{V} \leftarrow \{v_{i_1}, .., v_{i_{|V|}}\}$ such that $\sigma(v_{i_t}) < \sigma(v_{i_{t+1}})$
3 Initialize sequence $S \leftarrow \emptyset$
4 for $v \in \overline{V}$ do
5     Add $\Omega(v)$ at the end of $S$
6 end
7 return $S$
```

operation as 1D Convolutional Neural Network on the columns of this matrix. On top of LCNN is a fully-connected layer for regression tasks or classification tasks.

2.4.3. Gated Recurrent Unit / Long Short-Term Memory

Long Short-Term Memory (LSTM), firstly proposed by Schmidhuber and colleagues in 1997, is a special kind of Recurrent Neural Network that was designed for learning sequential and time-series data [10]. LSTM is widely applied into many current state-of-the-art Deep Learning models in various aspects of Machine Learning including Natural Language Processing, Speech Recognition and Computer Vision. Gated Recurrent Unit (GRU) was introduced by Bengio and colleagues in their NIPS 2014 paper in the context of sequential modeling [11]. GRU can be understood as a simplication of LSTM.

With the spirit of Language Modeling, throughout the neural network, from level 0 to level T, all representations of a vertex v can be represented as a sequence:

$$f_0(v) \rightarrow f_1(v) \rightarrow .. \rightarrow f_T(v)$$

in which $f_l(v)$ is more global than $f_{l-1}(v)$, and $f_{l-1}(v)$ is more local than $f_l(v)$. One can think of the sequence of representations as a sentence of words as in Natural Language Processing. We can embed GRU / LSTM at each level of our network in the sense that GRU / LSTM at level l will learn to choose whether to select $f_l(v)$ as the final representation or reuse one of the previous level representations $\{f_0(v), .., f_{l-1}(v)\}$. This idea, inherited from Gated Graph Sequence Neural Networks of Li and colleagues in ICLR 2016 [11], is well embeded and perfectly fit for our Covariant Compositional Networks model.

3. Covariant Compositional Networks

This section briefly discusses the framework of Covariant Composition Networks. For the full detail, readers are encouraged to read our ICLR 2018 workshop paper *Covariant Compositional Networks For Learning Graphs* [20].

3.1. Generic Algorithm

Covariant Compositional Networks are designed to have a hierarchical and multi-scale structure with multiple levels / layers to capture the structure of the input graph from local scale to global scale in such a way that representations of higher levels are built based on representations of lower levels and importantly respect permutation and rotational invariance. First of all, we define the hierarchical receptive field $\Omega_l(v)$ of a vertex v at level l recursively:

- $\Omega_0(v) = \{v\}$
- $\Omega_l(v) = \Omega_{l-1}(v) \cup \bigcup_{w \in B(v,1)} \Omega_{l-1}(w)$

The receptive field can be considered as the set of vertices centered at v. Information of v at level l will be collected from $\Omega_l(v)$ via generalized message passing scheme. Based on the definition of $\Omega_l(v)$, we generalize the vertex represetation $f_l(v)$ to higher-order representations. The zeroth order representation limits $f_l(v)$ to be a vector of d channels as discussed in section 2.4.

The first-order representation allows each channel of d channels of $f_l(v)$ to be represented by a vector of size $|\Omega_l(v)|$ in which each element of this vector corresponds to a vertex in the receptive field $\Omega_l(v)$. Thus in the first-order, $f_l(v) \in \mathbb{R}^{|\Omega_l(v)| \times d}$ where each row of $f_l(v)$ is a zeroth order representation of d channels of a vertex in $\Omega_l(v)$ at level $l-1$.

More general, the second order representation allows each channel of d channels of $f_l(v)$ to be represented by a symmetric matrix of size $|\Omega_l(v)| \times |\Omega_l(v)|$. Thus, $f_l(v) \in |\Omega_l(v)| \times |\Omega_l(v)| \times d$ is a 3rd order tensor.

The first-order aggregation rule can be defined as follows:

$$\phi_l^{first}(v) = \sum_{w \in \Omega_l(v)} X_l(v, w) f_{l-1}(w)$$

where $f_{l-1}(w) \in \mathbb{R}^{|\Omega_{l-1}(w)| \times d}$ for each $w \in \Omega_l(v)$, and $X_l(v, w) \in \{0, 1\}^{|\Omega_l(v)| \times |\Omega_{l-1}(w)|}$ is a permutation matrix defined as follows:

- $X_l(v, w)_{ij} = 1$: if $\Omega_l(v)_i = \Omega_{l-1}(w)_j$
- $X_l(v, w)_{ij} = 0$: otherwise

This permutation matrix arranges vertices in $\Omega_{l-1}(w)$ into the correct position in $\Omega_l(v)$. Remark that $\Omega_{l-1}(w) \subseteq \Omega_l(v)$. Obviously, the first-order aggregation rule give us the first-order representation $\phi_l^{first}(v) \in \mathbb{R}^{|\Omega_l(v)| \times d}$. The second-order aggregation rule can be defined as follows:

$$\phi_l^{second}(v) = \sum_{w \in \Omega_l(v)} X_l(v, w) \otimes f_{l-1}(w) \otimes X_l(v, w)^T$$

where $f_{l-1}(w) \in \mathbb{R}^{|\Omega_{l-1}(w)| \times |\Omega_{l-1}(w)| \times d}$ for each w, operatior $\otimes$ is the broad-casting matrix-tensor multiplication, and $X_l(v, w) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_{l-1}(w)|}$ is the permutation matrix defined as above. The second-order aggregation rule gives us the second-order representation $\phi_l^{second}(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times d}$.

The generic learning rule can be expressed as:

$$f_l(v) = \sigma\big(b_l + W_l \otimes \phi_l(v)\big)$$

where $\phi_l(v)$ can be obtained by zeroth-order, first-order or second-order aggregation rules; learnable weight matrix $W_l \in \mathbb{R}^{d \times d}$; learnable bias vector $b_l \in \mathbb{R}^d$; operator $\otimes$ represents broad-casting matrix-tensor multiplication in the sense that we apply an affine transformation to the d channels of $\phi_l(v)$; and σ is a non-linearity function. In our case, we choose the non-linearity as Leaky ReLU. We can see that $f_l(v)$ has the same number of channels as in the previous level, but in practice we can reduce the number of channels by half after each level to increase the robustness of the whole network, in particular $W_l \in \mathbb{R}^{\lfloor d/2 \rfloor \times d}$ and $b_l \in \mathbb{R}^{\lfloor d/2 \rfloor}$.

Suppose that the network has T levels. On the top level, for an input graph of $|V|$ vertices, we obtain $|V|$ tensors $\{f_T(v_1), .., f_T(v_{|V|})\}$. For each tensor, we *shrink* it into a d-dimensional vector by summation such that we get the set of $|V|$ d-dimensional vectors $\{\hat{f}_T(v_1), .., \hat{f}_T(v_{|V|})\}$. The graph G is represented by:

$$f_T(G) = \sum_{v \in V} \hat{f}_T(v)$$

in which each channel of $f_T(G)$ is a scalar. Based on $f_T(G)$, we can apply a fully-connected layer (linear regression, softmax or multi-layer perceptron) for regression or classification tasks.

3.2. Tensor Stacking

To empower the first-order and second-order representations, instead of summing up the lower-level vertex representations as in section 2.4. Generic Algorithm, we stack them into a higher-order tensor:

$$\phi_l^{first}(v) = \Phi\big\{X_l(v, w) f_{l-1}(w) \bigm| w \in \Omega_l(v)\big\}$$

$$\phi_l^{second}(v) = \Phi\big\{X_l(v, w) \otimes f_{l-1}(w) \otimes X_l(v, w)^T \bigm| w \in \Omega_l(v)\big\}$$

where $\Phi\{.\}$ denotes the tensor stacking operation. Thus, the first-order aggregation rule returns a 3rd order tensor:

$$\phi_l^{first}(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times d}$$

and similarly the second-order aggregation rule returns a 4th order tensor:

$$\phi_l^{second}(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times d}$$

3.3. Tensor Product

To capture more structure of graph, we introduce the tensor products between the aggregated representation with the reduced Adjacency matrix:

$$\hat{\phi}_l^{second}(v) \leftarrow \phi_l^{second}(v) \otimes A_{\Omega_l(v)}$$

where $A_{\Omega_l(v)} \in \{0, 1\}^{|\Omega_l(v)| \times |\Omega_l(v)|}$ such that $A_{\Omega_l(v)}(i, j) = 1$ if and only if there is an edge between two vertices $\Omega_l(v)_i$ and $\Omega_l(v)_j$. The operator $\otimes$ denotes a tensor product operation resulting into a 6th order tensor:

$$\hat{\phi}_l^{second}(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times d}$$

For notation simplicity, we denote $\hat{\phi}_l^{second}(v)$ as $\phi_l^{second}(v)$ with an understanding that $\phi_l^{second}(v)$ is obtained by tensor stacking first and then tensor product with the reduced adjacency matrix.

Instead of tensor product with the reduced adjacency matrix $A_{\Omega_l(v)}$, we can replace $A_{\Omega_l(v)}$ by the normalized graph Laplacian restricted to $\Omega_l(v)$, formally $\mathcal{L}_{\Omega_l(v)} = I_{|\Omega_l(v)|} - D^{-1/2}_{\Omega_l(v)} A_{\Omega_l(v)} D^{-1/2}_{\Omega_l(v)}$ where I is the identity matrix and D is the diagonal matrix of vertex degree, or the Coulomb matrix in some physical applications.

3.4. Virtual Indexing System

One of the most challenging problem is dealing with extremely high order tensors:

$$\phi_l^{first}(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times d}$$

$$\phi_l^{second}(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times d}$$

We cannot directly store these huge tensors in the memory. We need to emphasize that this problem is very difficult both algorithmically and systematically. For example, if the receptive field $\Omega_l(v)$ has 10 vertices and the number of channels $d = 10$, then to store $\phi_l^{first}(v)$ we need 10^3 floating-point numbers, and to store $\phi_l^{second}(v)$ we need 10^6 floating-point numbers. Moreover, the graph has $|V|$ vertices, each vertex v need that much floating-point numbers. We also have to take into account that the neural network has T levels, at each level we have to construct the representation for each vertex. Approximately the amount of memory is $O(T \times |V| \times |\Omega_l(v)|^5 \times d)$. It is obviously infeasible with all current computers.

We propose our solution **Virtual Indexing System** inspired by Virtual Machine idea in Operating Systems. One observation is that these huge tensors are very sparse and easy to compute component-wise. Instead of explictly stacking lower-order tensors into a higher-order one, we only keep the list of pointers pointing to these lower-order tensors. When we want to get a value of the result tensor, we can easily search the corresponding value via the list of pointers. Moreover, instead of explicitly computing the tensor product with the reduced adjacency matrix, we can just store the pointer to the reduce adjacency matrix. When we want the value at position indexed by (a, b, c, d, e, f), we search for the value at position indexed by (a, b, c, f) of the stacked tensor and the value at position indexed by (d, e) of the reduced adjacency matrix, and then multiply these two values. Remark that to access (a, b, c, f) of the stacked tensor, we need to go through the list of pointers as explained above.

The memory gain is significant, we only need $O(1)$ memory space for both tensor stacking and tensor product operations. The running time is proportional to the number of accesses to the result tensor.

3.5. Tensor Contraction

The tensor contraction operation is defined as a reduction from high-order tensors into low-order tensors that respects symmetry and permutation invariance. Formally, for the first-order, the tensor contraction is defined as a function:

$$\mathcal{F} : \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)|} \rightarrow \mathbb{R}^{|\Omega_l(v)|}$$

and for the second-order, the tensor contraction is defined as a function:

$$\mathcal{S} : \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)|} \rightarrow \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)|}$$

Again, we cannot implement the tensor contraction explicitly but via **Virtual Indexing System**. We apply the tensor contraction for each of d channels separately and then concatenate the result tensors. By combinatorics, for the first-order, there are 2 unique ways of contractions: sum of all elements, and the trace. In addition, one can introduce the Hadamard-type contraction as taking the diagonal. By combinatorics and symmetry of indices, for the second-order, there are exactly 18 unique ways of contractions. In conclusion, after tensor stacking, tensor product (for the second-order only) and tensor contraction, the first-order representation is:

$$\phi_l^{first}(v) \in \mathbb{R}^{|\Omega_l(v)| \times 2d}$$

and the second-order representation is:

$$\phi_l^{second}(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times 18d}$$

The sizes are very manageable now. We apply the learnable matricies $W_l^{first} \in \mathbb{R}^{d \times 2d}$ and $W_l^{second} \in \mathbb{R}^{d \times 18d}$ to avoid exponentially growing number of channels, in detail:

$$\phi_l^{first}(v) \cdot (W_l^{first})^T \in \mathbb{R}^{|\Omega_l(v)| \times d}$$

$$W_l^{second} \otimes \phi_l^{second}(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times d}$$

The 18 unique contractions can be implemented with Virtual Indexing System as in algorithm 7.

Algorithm 7: Virtual tensor contraction

```
Data: Set of $N$ third-order tensors $\mathcal{X} = \{x^1, .., x^N\}$
      such that $x^i \in \mathbb{R}^{N \times N \times c}$, and an adjacency matrix
      $A \in \mathbb{R}^{N \times N}$. Given indices $(a, b, c, d, e, f)$,
      return $x^a_{b,c,f} \cdot A_{d,e}$ as the tensor value $\mathcal{T}_{a,b,c,d,e,f}$
      to avoid holding a sixth-order tensor in memory.
Result: $\phi \in \mathbb{R}^{N \times N \times (18c)}$
1 Five $(1+1+1)$ cases: Compute $\{\phi^1, .., \phi^5\}$
2 Ten $(1+2)$ cases: Compute $\{\phi^6, .., \phi^{15}\}$
3 Three $(3)$ cases: Compute $\{\phi^{16}, \phi^{17}, \phi^{18}\}$
4 $\phi \leftarrow \bigoplus_{i=1}^{18} \phi^i$ where $\phi^i \in \mathbb{R}^{N \times N \times c}$
5 return $\phi$
```

3.6. First-order CCNs

To describe the first-order CCNs algorithm, we define operation $\Phi(\mathcal{X})$ as stacking same-size matrices $x \in \mathcal{X}$ to a 3rd order tensor. Let $\Theta(x)$ as summing up all rows of x, and reshape it to a column vector. Finally, we define the permutation matrix $\chi(\Omega_l(v), \Omega_{l-1}(u)) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_{l-1}(u)|}$ as

$$\chi(\Omega_l(v), \Omega_{l-1}(u))_{ij} \triangleq \begin{cases} 1, & \text{if } \Omega_l(v)_i = \Omega_{l-1}(u)_j \\ 0, & \text{otherwise} \end{cases}$$

Remark that the number of contractions is 2. Contruction of the neural network is descibed with pseudocode 8.

3.7. Second-order CCNs

We define $\Theta(x)$ as shrinking a tensor x to a column vector of c channels, and $\otimes$ as broadcast multiplication. Remark that based on the symmetry, we have exactly 18 unique contractions. Contruction of the neural network is described with pseudocode 9.

4. GraphFlow Deep Learning Framework

4.1. Motivation

Many Deep Learing frameworks have been proposed over the last decade. Among them, the most successful ones are TensorFlow [12], PyTorch [13], Mxnet [15], Theano [14]. However, none of these frameworks are completely suitable for graph neural networks in the domain of molecular applications with high complexity tensor operations due to the following reasons:

Algorithm 8: First-order CCNs

```
Data: Input graph $G = (V, E)$. Input vertex feature
      vector $l_v \in \mathbb{R}^c$. Receptive field $\Omega_l(v)$. Feature
      tensor $f_v^l$. Learnable weight matrices $W^l$.
      Non-linearity function $\sigma$. Layers indexed from 0
      to $L$.
1  for $v \in V$ do
2      $\Omega_0(v) \leftarrow \{v\}$
3      $f_v^0 \leftarrow \sigma(W^0 l_v)$ where $W^0 \in \mathbb{R}^{c \times c}$
4      Reshape $f_v^0$ to $1 \times c$
5  end
6  $f^0 \leftarrow \sum_{v \in V} \Theta(f_v^0)$
7  for $l = 1 \to L$ do
8      for $v \in V$ do
9          $\Omega_l(v) \leftarrow \bigcup_{u \in \mathcal{N}(v)} \Omega_{l-1}(u)$
10         $\chi \leftarrow \chi(\Omega_l(v), \Omega_{l-1}(u))$
11         $\hat{\phi}_v^l \leftarrow \Phi(\{\chi \cdot f_u^{l-1} | u \in \Omega_l(v)\})$
12         Reduce $\hat{\phi}_v^l \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times c}$ to
             $\phi_v^l \in \mathbb{R}^{|\Omega_l(v)| \times (2c)}$
13         $f_v^l \leftarrow \sigma(\phi_v^l W^l)$ where $W^l \in \mathbb{R}^{(2c) \times c}$
14     end
15     $f^l \leftarrow \sum_{v \in V} \Theta(f_v^l)$
16 end
17 Graph feature $f \leftarrow \bigoplus_{l \in [0,L]} f^l \in \mathbb{R}^{(L+1)c}$
18 Use $f$ for downstream tasks
```

Algorithm 9: Second-order CCNs

```
1  for $v \in V$ do
2      $\Omega_0(v) \leftarrow \{v\}$
3      $f_v^0 \leftarrow \sigma(W^0 l_v)$ where $W^0 \in \mathbb{R}^{c \times c}$
4      Reshape $f_v^0$ to $1 \times 1 \times c$
5  end
6  $f^0 \leftarrow \sum_{v \in V} \Theta(f_v^0)$
7  for $l = 1 \to L$ do
8      for $v \in V$ do
9          $\Omega_l(v) \leftarrow \bigcup_{u \in \mathcal{N}(v)} \Omega_{l-1}(u)$
10         for $u \in \Omega_l(v)$ do
11             $\chi \leftarrow \chi(\Omega_l(v), \Omega_{l-1}(u))$
12             $\phi_{v,u}^l \leftarrow \chi \otimes f_u^{l-1} \otimes \chi^T \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times c}$
13         end
14         Apply virtual tensor contraction algorithm 7
             with inputs $\{\phi_{v,u}^l | u \in \Omega_l(v)\}$ and the
             restricted adjacency matrix $A \downarrow_{\Omega_l(v)}$ to
             compute $\phi_v^l \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times (18c)}$.
15         $f_v^l \leftarrow \sigma(\phi_v^l \otimes W^l)$ where $W^l \in \mathbb{R}^{(18c) \times c}$
16     end
17     $f^l \leftarrow \sum_{v \in V} \Theta(f_v^l)$
18 end
19 Graph feature $f \leftarrow \bigoplus_{l \in [0,L]} f^l \in \mathbb{R}^{(L+1)c}$
20 Use $f$ for downstream tasks
```

- All of these frameworks do not support tensor contractions and other sophisticated tensor operations. Moreover, they are not flexible for an implementation of the Virtual Indexing System for efficient and low-cost tensor operations.
- The most widely used Deep Learning framework - TensorFlow is incapable of constructing dynamic computation graphs during run time that is essential for graph neural networks which are dynamic in size and structure. To get rid of static computation graphs, Google Research has been proposed an extension of TensorFlow that is TensorFlow-fold but has not completely solved the flexibility problem [16].

To address all these drawbacks, we implement from scratch our **GraphFlow Deep Learning Framework** in C++11 with the following criteria:

1. Supports symbolic / automatic differentiation that allows users to construct any kind of neural networks without explicitly writing the complicated back-propagation code each time.
2. Supports dynamic computation graphs that is fundamental for graph neural networks such that partial computation graph is constructed before training and the rest is constructed during run time depending on the size and structure of the input graphs.
3. Supports sophisticated tensor / matrix operations with Virtual Indexing System.
4. Supports tensor / matrix operations implemented in CUDA for computation acceleration by GPUs.

4.2. Overview

GraphFlow is designed with the philosophy of Object Oriented Programming (OOP). There are several classes divided into the following groups:

1. **Data structures**: `Entity`, `Vector`, `Matrix`, `Tensor`, etc. Each of these components contain two arrays of floating-point numbers: `value` for storing the actual values, `gradient` for storing the gradients (that is the partial derivative of the loss function) for the purpose of automatic differentiation. Also, in each class, there are two functions: `forward()` and `backward()` in which `foward()` to evaluate the network values and `backward()` to compute the gradients. Based on the OOP philosophy, `Vector` inherits from `Entity`, and both `Matrix` and `Tensor` inherit from `Vector`, etc. It is essentially important because polymorphism allows us to construct the computation graph of the neural network as a Directed Acyclic Graph (DAG) of `Entity` such that `forward()` and `backward()` functions of different classes can be called with object casting.
2. **Operators**: Matrix Multiplication, Tensor Contraction, Convolution, etc.

   For example, the matrix multiplication class `MatMul` inherits from `Matrix` class, and has 2 constructor parameters in `Matrix` type. Suppose that we have an object `A` of type `MatMul` that has 2 `Matrix` inputs `B` and `C`. In the `forward()` pass, `A` computes its value as `A = B * C` and stores it into `value` array. In the `backward()` pass, `A` got the gradients into `gradient` (as flowing from the loss function) and increase the gradients of `B` and `C`.

   It is important to note that our computation graph is DAG and we find the topological order to evaluate `value` and `gradient` in the correct order. That means `A -> forward()` is called after both `B -> forward()` and `C -> forward()`, and `A -> backward()` is called before both `B -> backward()` and `C -> backward()`.
3. **Optimization algorithms**: Stochastic Gradient Descent (SGD), SGD with Momentum, Adam, AdaGrad, AdaMax, AdaDelta, etc. These algorithms are implemented into separate drivers: these drivers get the values and gradients of learnable parameters computed by the computation graph and then optimize the values of learnable parameters algorithmically.
4. **Neural Networks objects**: These are classes of neural network architectures implemented based on the core of GraphFlow including graph neural networks (for example, CCN, NGF and LCNN), convolutional neural networks, recurrent neural networks (for example, GRU and LSTM), multi-layer perceptron, etc. Each class has multiple supporting functions: load the trained learnable parameters from files, save them into files, learning with mini-batch or without mini-batch, using multi-threading or not, etc.

The figure of GraphFlow is contained in the Appendix.

4.3. Parallelization

4.3.1. Efficient Matrix Multiplication In GPU

Multiple operations of a neural network can be expressed as matrix multiplication. Having a fast implementation of matrix multiplication is extremely important for a Deep Learning framework. We have implemented two versions of matrix multiplication in CUDA: one using naive kernel function that accesses matrices directly from the global memory of GPU, one is more sophisticated kernel function that uses shared memory in which the shared memory of each GPU block contains 2 blocks of the 2 input matrices to avoid latency of reading from the GPU global memory. Suppose that each GPU block can execute up to 512 threads concurrently, we select the block size as 22 x 22. The second approach outperforms the first approach in our stress experiments.

4.3.2. Efficient Tensor Contraction In CPU

Tensor stacking, tensor product, tensor contraction play the most important role in the success of Covariant Compositional Networks. Among them, tensor contraction is the most difficult operation to implement efficient due to the complexity of its algorithm. Let consider the second-order tensor product:

$$\phi_l(v) \otimes A_{\Omega_l(v)}$$

where $\phi_l(v) \in \mathbb{R}^{|\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times d}$ is the result from tensor stacking operation of vertex v at level l, and $A_{\Omega_l(v)} \in \{0,1\}^{|\Omega_l(v)| \times |\Omega_l(v)|}$ is the restricted adjacency matrix to the receptive field $\Omega_l(v)$. With the Virtual Indexing System, we do not compute the full tensor product result, indeed we compute some elements of it when necessary.

The task is to contract / reduce the tensor product $\phi_l(v) \otimes A_{\Omega_l(v)}$ of 6th order into 3rd order tensor of size $|\Omega_l(v)| \times |\Omega_l(v)| \times d$. As discussed section 3.5, because of symmetry, there are exactly 18 unique ways of contractions in the second-order case. Suppose that our CPU has $N < 18$ cores, assuming that we can run all these cores concurrently, we launch N threads such that each thread processes $\lceil 18/N \rceil$ contractions. There can be some threads doing more or less contractions.

One challenge is about synchronization: we have to ensure that the updating operations are atomic ones.

4.3.3. Efficient Tensor Contraction In GPU

The real improvement in performance comes from GPU. Thus, in practice, we do not use the tensor contraction with multi-threading in CPU. Because we are experimenting on Tesla GPU K80, we have an assumption that each block of GPU can launch 512 threads and a GPU grid can execute 8 concurrent blocks. In GPU global memory, $\phi_l(v)$ is stored as a float array of size $|\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times d$, and the reduced adjacency matrix $A_{\Omega_l(v)}$ is stored as a float array of size $|\Omega_l(v)| \times |\Omega_l(v)|$. We divide the job to GPU in such a way that each thread processes a part of $\phi_l(v)$ and a part of $A_{\Omega_l(v)}$. We assign the computation work equally among threads based on the estimated asymptotic complexity.

Again, synchronization is also a real challenge: all the updating operations must be the atomic ones. However, having too many atomic operations can slow down our concurrent algorithm. That is why we have to design our GPU algorithm with the minimum number of atomic operations as possible. We obtain a much better performance with GPU after careful consideration of all factors.

4.3.4. CPU Multi-threading In Gradient Computation

Given a minibatch of M training examples, it is a natural idea that we can separate the gradient computation jobs into multiple threads such that each thread processes exactly one training example at a time before continuing to process the next example. We have to make sure that there is no overlapping among these threads. After completing the gradient computations from all these M training examples, we sum up all the gradients, average them by M and apply an variant of Stochastic Gradient Descent to optimize the neural networks before moving to the next minibatch.

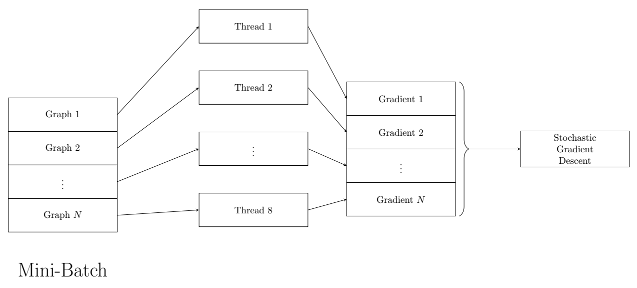
Technically, suppose that we can execute T threads concurrent at a time for gradient computation jobs. Before every training starts, we initialize exactly T identical dynamic computation graphs by GraphFlow. Given a minibatch of M training examples, we distribute the examples to T thread, each thread uses a different dynamic computation graph for its gradient computation job. By this way, there is absolutely no overlapping and our training is completely synchronous.

The minibatch training with CPU multi-threading is described by figure 2.

4.3.5. Reducing data movement between GPU and main memory

One challenge of using GPU is that we must move the data from the main memory to the GPU before performing any

Figure 2: *CPU multi-threading for gradient computation*



computation. This could prevent us from achieving high efficient computation since data movement takes time and the GPU cannot be used until this process completes. We solve this problem by detaching data movement from computation. We introduce two new functions, `upload` and `download`, let them handle the data movement to and from the GPU, respectively. The `forward` and `backward` functions only perform computation. This approach makes the framework more flexible as it can dynamically determine which parts of the data should be moved and when to move them. Therefore, the framework has more options when scheduling computation flows of the network, enabling better GPU utilization as well as avoiding unnecessary communication caused by poor implementations.

Figure 3 shows an example of our approach. On the left of the figure is the C++ for computing matrix computation of $A \times B \times C$. The execution of the code is depicted on the right side. Firstly, the framework copies necessary data from main memory to GPU's global memory by calling upload function. The result and gradient is then generated after the calls to `forward()` and `backward()` functions. Those functions work in the way similar to what we introduced in previous sections except that the computation is performed entirely by the GPU. In the end, `download` function is called to move the computation results back to the main memory. Clearly, data movement occurs only during the initialization and finalization of the whole process, there is no communication between GPU and main memory during the computation so the performance would be improved significantly.

4.3.6. Source code

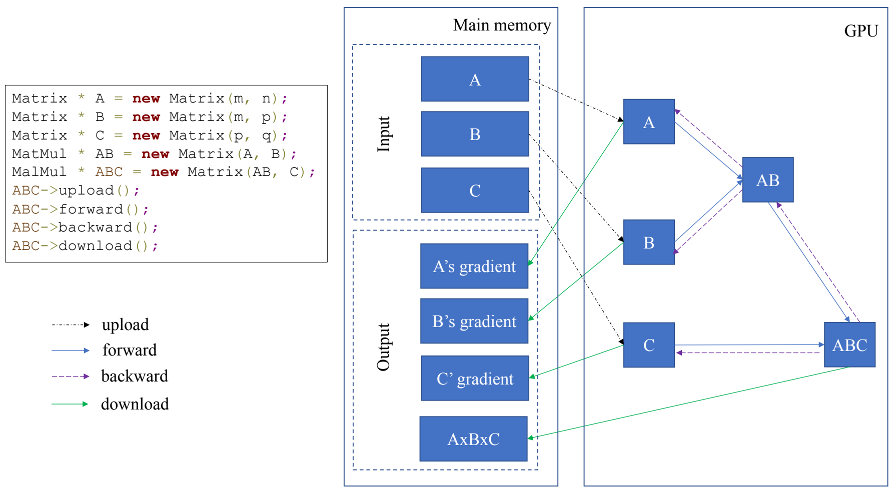
The source code of GraphFlow Deep Learning framework can be found at `https://github.com/HyTruongSon/GraphFlow`.

5. Experiments and Results

5.1. HCEP Dataset

The Harvard Clean Energy Project (HCEP) dataset [2] contains 2.3 million molecules that are potential future solar energy materials. By expensive Density Functional Theory, the authors of HCEP computed the Power Conversion Energy (PCE) for each molecule. PCE values are continuous ranging from 0 to 11. For our experiments, we extract 50,000 molecules and set 30,000 molecules for training, 10,000 molecules for validating, and 10,000 molecules for testing. Each molecule is represented

Figure 3: *Example of data flow between GPU and main memory*



```
Matrix * A = new Matrix(m, n);
Matrix * B = new Matrix(m, p);
Matrix * C = new Matrix(p, q);
MatMul * AB = new Matrix(A, B);
MalMul * ABC = new Matrix(AB, C);
ABC->upload();
ABC->forward();
ABC->backward();
ABC->download();
```

Figure 4: *WL feature vectors with PCA and t-SNE*

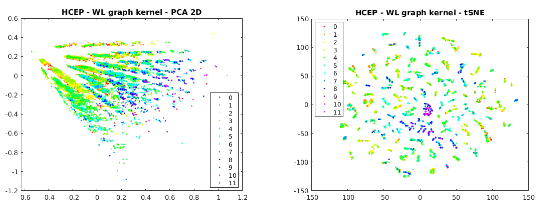


Figure 5: *CCN 1D feature vectors with PCA and t-SNE*

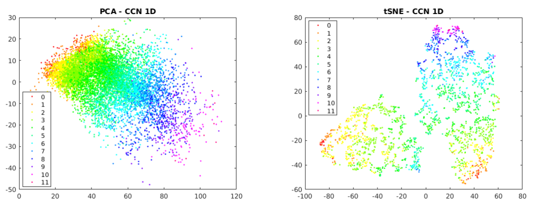
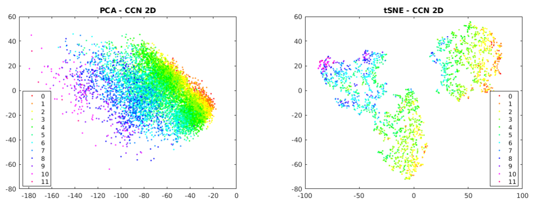


Figure 6: *CCN 2D feature vectors with PCA and t-SNE*



by a SMILES code. We transform the SMILES codes into adjacency matrices where each atom is a vertex and the atom type is the input vertex label.

For the Dictionary Weisfeiler-Lehman graph kernel, we did an investigation on the number of different receptive fields. In the set of 50,000 molecules that we selected, there are in total 11,172 different receptive fields of level 3. This number is very manageable as the size of the dictionary. We also applied *sparse* Support Vector Regression to these sparse feature vectors and obtained good results.

5.1.1. Visualization

For the purpose of visualization, we rounded PCE values into the nearest integers. We applied Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) [8] on the feature vectors produced by both Weisfeiler-Lehman (WL) graph kernel and our Covariant Compositional Networks.

We observed that our CCN models give a much better visualization in 2-dimensional visualization. We can see the separations among the clusters where each cluster is associated with a rounded integer PCE value.

5.1.2. Regression tasks

Regarding Weisfeiler-Lehman (WL) graph kernel, we apply Linear regression, Gaussian Process and SVM for the task of regressing PCE values. We find the optimal hyperparameters in the validation set. For linear regression, the regularization parameter $C \in \{0, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10, 100\}$. For Gaussian Process, we choose Radial Basis Function (RBF) kernel with the optimal $\sigma = 2$, the noise parameter is selected from the range from 0 to 100. For SVM, the RBF kernel is chosen with $\sigma = 0.01$, and regualarization parameter $C = 100$.

One of the baseline is dK-Series method that was first proposed by [25]. The idea of dK-Series can be summarized as:

1. Extract all subgraphs of size d in a molecular graph.
2. Classify each subgraph as 1 element of the set of non-isomorphic graphs of size d.
3. Based on the subgraph classification, we build the frequency vector (probability distribution) and use the frequency vector as the molecular graph representation.

Remark that for our molecular application, instead of using vertex degree, we use atomic types (for example, Carbon - C, Hydrogen - H, etc.). We experimented with

Table 1: *HCEP dataset regression results*

Model	Test MAE	Test RMSE
dK-Series $d = 1$	1.492	3.457
dK-Series $d = 2$	1.255	2.545
dK-Series $d = 3$	1.130	2.194
dK-Series $d = 4$	1.025	1.774
WL + Linear Regression	0.805	1.096
WL + Gaussian Process	0.760	1.093
WL + Support Vector Machine	0.878	1.202
NGF	0.851	1.177
LCNN	0.718	0.973
CCN 1D	0.216	0.291
CCN 2D	0.340	0.449

different values of $d \in \{1, 2, 3, 4\}$. Our implementation of dK-Series on HCEP dataset can be found at `https://github.com/HyTruongSon/dk-series`.

Regarding training our Covariant Compositional Networks, for the first-order representation (CCN 1D), we select the best architecture among the number of levels / layers 1, 2, 3, 4, 5, 6, and 7. CCN 1D gives the best result with 7 levels. For the second-order representation (CCN 2D), the number of levels / layers is 1 and 2. CCN 2D gives the best result with a single level. The number of channels is fixed as 10. We employ Stochastic Gradient Descent with Adam optimization. The learning rate is initially 10^{-3} and linearly decayed to 10^{-6} in 10^6 iterations. The batch size is fixed as 64. The maximum number of epochs is 1024 in all settings.

We also compare with two other state-of-the-art graph neural networks: Neural Graph Fingerprint (NGF) and Learning Convolutional Neural Networks (LCNN). NGF has the maximum of 3 levels and the number of hidden states is 10. LCNN has 2 convolutional layers. In all graph neural networks settings, we always have a linear regression on top of the network.

We estimate the performance of our models in both Mean Average Error (MAE) and Root Mean Square Error (RMSE). In both metrics, Covariant Compositional Networks outperform Weisfeiler-Lehman graph kernel, Neural Graph Fingerprint and Learning Convolutional Neural Networks. We observe that trainings for higher-order representations are increasingly harder since the size of representation grows exponentially. Some techniques have been used to solve this problem, for example thresholding the maximum size of receptive fields. More training is probably required to obtain the full potential of higher-order representations.

5.2. QM9 dataset

QM9 [17] is a dataset of $\sim$ 134k organic molecules with up to nine heavy atoms (C,O,N and F) out of the GDB-17 universe [18] of molecules. Each molecule contains data including 13 target chemical properties, along with the spatial position of every constituent atom. With this dataset, we have executed 2 different experiments:

- **QM9(a)**: We predict the 13 target properties of every

Figure 7: *Testing errors vs Number of training epochs. Top: MAE. Down: RMSE.*

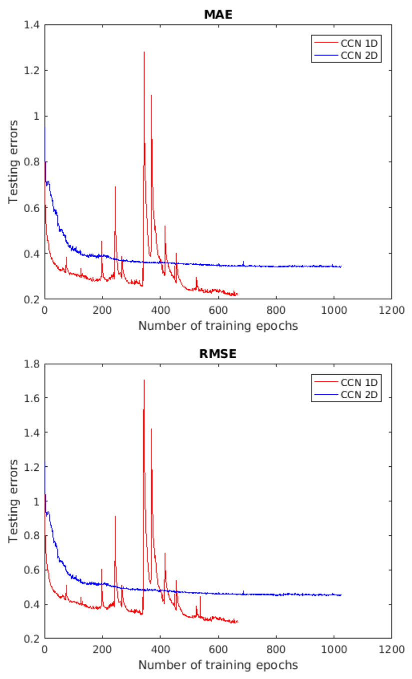


Figure 8: *Distributions of the predicted PCE values and the expected ones. Top: CCN 1D. Down: CCN 2D.*

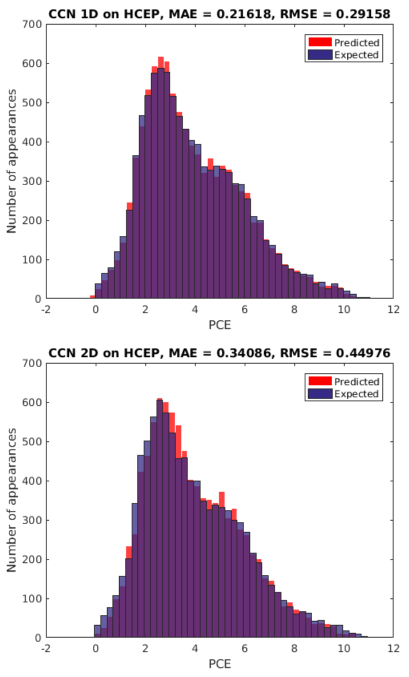


Table 2: *QM9(a) regression results (MAE)*

Target	WLGK	NGF	PSCN	CCN 2D
alpha	0.46	0.43	0.20	**0.16**
Cv	0.59	0.47	0.27	**0.23**
G	0.51	0.46	0.33	**0.29**
gap	0.72	0.67	0.60	**0.54**
H	0.52	0.47	0.34	**0.30**
HOMO	0.64	0.58	0.51	**0.39**
LUMO	0.70	0.65	0.59	**0.53**
mu	0.69	0.63	0.54	**0.48**
omega1	0.72	0.63	0.57	**0.45**
R2	0.55	0.49	0.22	**0.19**
U	0.52	0.47	0.34	**0.29**
U0	0.52	0.47	0.34	**0.29**
ZPVE	0.57	0.51	0.43	**0.39**

Table 3: *QM9(b) regression results (MAE)*

Target	CCNs	DFT error	Physical unit
alpha	**0.19**	0.4	Bohr^3
Cv	**0.06**	0.34	cal/mol/K
G	**0.05**	0.1	eV
gap	**0.11**	1.2	eV
H	**0.05**	0.1	eV
HOMO	**0.08**	2.0	eV
LUMO	**0.07**	2.6	eV
mu	0.43	**0.1**	Debye
omega1	**2.54**	28	cm^{-1}
R2	5.03	-	Bohr^2
U	**0.06**	0.1	eV
U0	**0.05**	0.1	eV
ZPVE	**0.0043**	0.0097	eV

molecule. For this text we consider only heavy atoms and exclude hydrogen. Vertex feature initialization is performed in the same manner as the HCEP experiment. For training the neural networks, we normalized all 13 learning targets to have mean 0 and standard deviation 1. We report the MAE with respect to the normalized learning targets.

- **QM9(b)**: The QM9 dataset with each molecule including hydrogen atoms. We use both physical atomic information (vertex features) and bond information (edge features) including: atom type, atomic number, acceptor, donor, aromatic, hybridization, number of hydrogens, Euclidean distance and Coulomb distance between pair of atoms. All the information is encoded in a vectorized format.

  To include the edge features into our model along with the vertex features, we used the concept of a *line graph* from graph theory. We constructed the line graph for each molecular graph in such a way that: an edge of the molecular graph corresponds to a vertex in its line graph, and if two edges in the molecular graph share a common vertex then there is an edge between the two corresponding vertices in the line graph. The edge features become vertex features in the line graph. The inputs of our model contain both the molecular graph and its line graph. The feature vectors F_ℓ between the two graphs are merged at each level ℓ.

  In QM9(b), we report the mean average error for each learning target in its corresponding physical unit and compare it against the Density Functional Theory (DFT) error given by [19].

5.3. Discussion

On the subsampled HCEP dataset, CCN outperforms all other methods by a very large margin. In the QM9(a) experiment, CCN obtains better results than three other graph learning algorithms for all 13 learning targets. In the QM9(b) experiment, our method gets smaller errors comparing to the DFT calculation in 11 out of 12 learning targets (we do not have the DFT error for R2).

6. Conclusion and Future Research

We extended the state-of-the-art Weisfeiler-Lehman graph kernel and generalized convolution operation for Covariant Compositional Networks by higher-order representations in order to approximate Density Functional Theory. We obtained very promising results and outperformed two other current state-of-ther-art graph neural networks such as Neural Graph Fingerprint and Learning Convolutional Neural Networks on Harvard Clean Energy Project and QM9 datasets. Thanks to parallelization, we significantly improved our empirical results.

We are developing our custom Deep Learning framework in C++ named **GraphFlow** which supports automatic and symbolic differentitation, dynamic computation graph as well as complex tensor / matrix operations in CUDA with GPU computation acceleration. We expect that this framework will enable us to design more flexible, efficient graph neural networks with molecular applications at a large scale in the future.

7. Acknowledgements

We would like to acknowledge Professor Risi Kondor for his valuable instruction and especially for his ideas for generalizing convolution operations in graphs. We also want to thank other members of Machine Learning group at the University of Chicago for their dedicated support.

Some of the neural network training in this paper was done using the Techstaff cluster of the Department of Computer Science and Midway cluster of the UChicago Computing Research Center.

8. References

[1] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, K. M. Borgwardt, “Weisfeiler-Lehman Graph Kernels”, *Journal of Machine Learning Research*, vol. 12, 2011.

[2] J. Hachmann, R. O. Amaya, S. A. Evrenk, C. A. Bedolla, R. S. S. Carrera, A. G. Parker, L. Vogt, A. M. Brockway, and A. A. Guzik, “The Harvard Clean Energy Project: Large-Scale Computational Screening and Design of Organic Photovoltaics on the World Community Grid”, *The Journal of Physical Chemistry Letters*, pp. 2241–2251, 2011.

[3] R. I. Kondor, and J. Lafferty, “Diffusion Kernels on Graphs and Other Discrete Structures”, *International Conference on Machine Learning (ICML)*, 2002.

[4] N. M. Kriege, P. L. Giscard, R. C. Wilson, “On Valid Optimal Assignment Kernels and Applications to Graph Classification”, *Neural Information Processing Systems (NIPS)*, 2016.

[5] S. Kearnes, K. McCloskey, M. Berndl, V. Pande, P. Riley, “Molecular Graph Convolutions: Moving Beyond Fingerprints”, *Journal of Computer-Aided Molecular Design*, vol. 30, pp. 595–608, 2016.

[6] D. Duvenaud, D. Maclaurin, J. A. Iparraguirre, R. G. Bombarelli, T. Hirzel, A. A. Guzik, R. P. Adams, “Convolutional Networks on Graphs for Learning Molecular Fingerprints”, *Neural Information Processing Systems (NIPS)*, 2015.

[7] T. N. Kipf, M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks”, *International Conference on Learning Representations (ICLR)*, 2017.

[8] L. V. D. Maaten, G. Hinton, “Visualizing Data using t-SNE”, *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[9] M. Niepert, M. Ahmed, K. Kutzkov, “Learning Convolutional Neural Networks”, *International Conference on Machine Learning (ICML)*, 2016.

[10] S. Hochreiter, J. Schmidhuber, “Long Short-Term Memory”, *Neural Computation*, 1997.

[11] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”, *Neural Information Processing Systems (NIPS) Workshop*, 2014.

[11] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, “Gated Graph Sequence Neural Networks”, *International Conference of Learning Representations (ICLR)*, 2016.

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”, *https://arxiv.org/abs/1603.04467*, 2016.

[13] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, “Automatic differentiation in PyTorch”, *Neural Information Processing Systems (NIPS)*, 2017.

[14] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. B. Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brbisson, O. Breuleux, P. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M. Cote, M. Cote, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. E. Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, “Theano: A Python framework for fast computation of mathematical expressions”, *https://arxiv.org/abs/1605.02688*, 2016.

[15] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems”, *Neural Information Processing Systems (NIPS) Workshop*, 2016.

[16] M. Looks, M. Herreshoff, D. Hutchins, P. Norvig, “Deep Learning with Dynamic Computation Graphs”, *International Conference of Learning Representations (ICLR)*, 2017.

[17] R. Ramakrishnan and P. O. Dral and M. Rupp and O. A. von Lilienfeld, “Quantum chemistry structures and properties of 134 kilo molecules”, *Sci. Data, vol. 1*, 2014.

[18] L. Ruddigkeit, R. van Deursen, L. C. Blum, Jean-Louis Reymond, “Enumeration of 166 Billion Organic Small Molecules in the Chemical Universe Database GDB-17”, *J. Chem. Inf. Model., vol. 52*, 2012.

[19] F. A. Faber, L. Hutchison, B. Huang, J. Gilmer, S. S. Schoenholz, G. E. Dahl, O. Vinyals, S. Kearnes, P. F. Riley, O. A. von Lilienfeld, “Prediction Errors of Molecular Machine Learning Models Lower than Hybrid DFT Error”, *J. Chem. Theory Comput., vol. 13*, 2017.

[20] R. Kondor, H. T. Son, H. Pan, B. Anderson, S. Trivedi, “Covariant Compositional Networks For Learning Graphs”, *International Conference of Learning Representation (ICLR) Workshop*, `https://arxiv.org/abs/1801.02144`, 2018.

[21] T. Weisfeiler, A. A. Lehman, “A reduction of a graph to a canonical form and an algebra arising during this reduction”, *Nauchno-Technicheskaya Informatsia*, 1968.

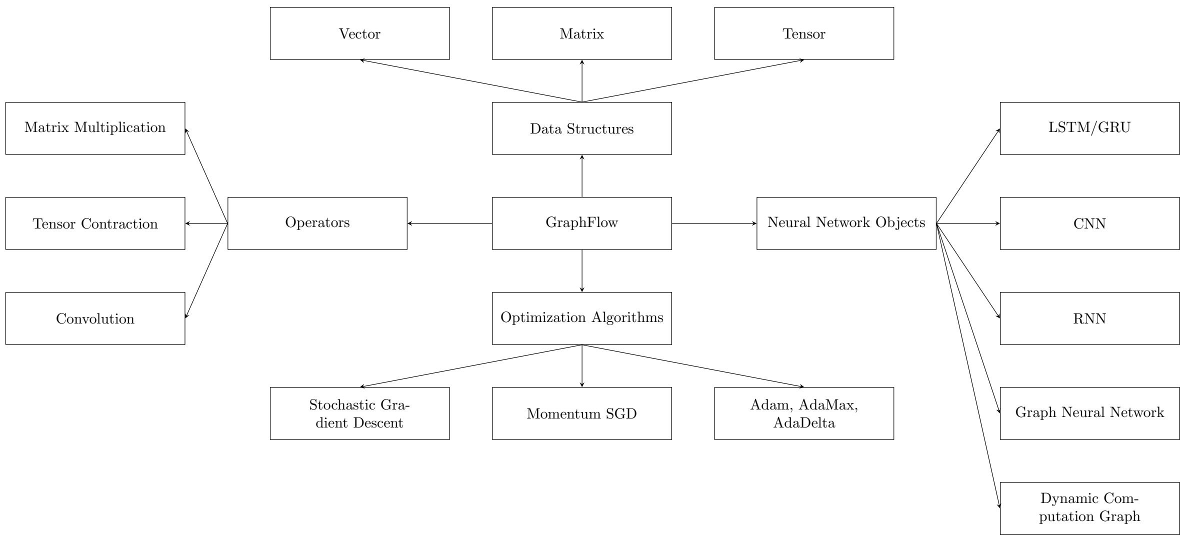
[22] D. J. C. Mackay, “Gaussian Processes: A Replacement for Supervised Neural Network?”, *Tutorial lecture notes for NIPS 1997*, 1997.

[23] C. J. C. Burges, “A tutorial on support vector machines for pattern recognition”, *Data Mining and Knowledge Discovery, vol. 2*, 1998.

[24] S. Mika, B. Schölkopf, A. Smola, K. B. Müller, M. Scholz, G. Rätsch, “Kernel PCA and De-Noising in Feature Spaces”, *Advances in Neural Information Processing Systems 11*, 1998.

[25] P. Mahadevan, D. Krioukov, K. Fall, A. Vahdat, “Systematic Topology Analysis and Generation Using Degree Correlations”, *ACM SIGCOMM*, 2006.

9. Appendix


Vector
Matrix
Tensor
Data Structures
Matrix Multiplication
Tensor Contraction
Convolution
Operators
GraphFlow
Neural Network Objects
LSTM/GRU
CNN
RNN
Graph Neural Network
Dynamic Computation Graph
Optimization Algorithms
Stochastic Gradient Descent
Momentum SGD
Adam, AdaMax, AdaDelta