Graph Neural Networks with Efficient Tensor Operations in CUDA/GPU and GraphFlow Deep Learning Framework in C++ for Quantum Chemistry

Hy Truong Son, Chris Jones

Department of Computer Science, The University of Chicago

hytruongson@uchicago.edu, csj@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 molecular dataset.

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

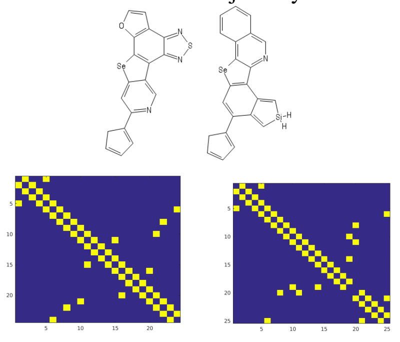
1. Introduction

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 is quadratic space usage and quadratic time-complexity in the number of examples. In this paper, we address this drawback by proposing Dictionary Weisfeiler-Lehman graph features in combination with Morgan circular fingerprints in section 3.1. 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. We detail the graph neural network baselines such as Neural Graph Fingerprint [6] and Learning Convolutional Neural Networks [9] in section 3.2. 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 zero-order as a vector to the first-order as a matrix and the second-order as a 3-order tensor in section 3.3. Also in this section, we introduce the notions of tensor contractions and tensor products to keep the orders of tensors manageable without exponentially growing. Our generalized convolution graph neural network is named as Covariant Compositional Networks.

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 chapter 4. We carefully evaluate the performance improvement of GraphFlow before and after parallelization in various experiments from testing tensor/matrix operations to training on a small molecular dataset. The performance gain is significant. Finally, we apply our methods to the Harvard Clean Energy Project (HCEP) molecular dataset [2]. The visualizations, experiments and empirical results are detailed in chapter 5. Chapter 6 is our conclusion and future research direction.

Figure 1. Two molecules with adjacency matrices in HCEP



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

2. Related works

The notion of graph kernel was first introduced by Kondor and Lafferty in their ICML 2002 paper [3]. Borgwardt and colleagues proposed the state-of-the-art Weisfeiler-Lehman graph kernel based on the famous Weisfeiler-Lehman test for graph isomorphism [1]. N. M. Kriege, P. L. Giscard and R. C. Wilson applied the Hungarian matching algorithm to define a more robust Weisfeiler-Lehman graph kernel [4]. The convolution on graphs as well as Graph Neural Network for learning molecular fingerprints are mentioned by other authors [5], [6], and [7].

3. Methods

3.1. Graph Kernel

3.1.1. Positive Semi-definite Kernel

Given two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Each vertex is associated with a feature vector $f : V \rightarrow \Omega$. A *positive semi-definite* graph kernel between G_1 and G_2 is defined as:

$$\mathcal{K}_{graph}(G_1, G_2) = \frac{1}{|V_1|} \cdot \frac{1}{|V_2|} \cdot \sum_{v_1 \in V_1} \sum_{v_2 \in V_2} k_{base}(f(v_1), f(v_2))$$

where k_{base} is the base kernel and can be:

- Linear: $k_{base}(x, y) = \langle x, y \rangle_{norm} = x^T y / (\|x\| \cdot \|y\|)$
- Quadratic: $k_{base}(x, y) = (\langle x, y \rangle_{norm} + q)^2$
- RBF: $k_{base}(x, y) = \exp(-\gamma \|x - y\|^2)$

3.1.2. Weisfeiler-Lehman Graph Feature

We combine Weisfeiler-Lehman graph kernel [1] and Morgan circular fingerprints into the Dictionary Weisfeiler-Lehman graph feature algorithm. To express the idea of this combination, we define a Weisfeiler-Lehman subtree level l rooted at a vertex v as the shortest-path subtree that includes all vertices reachable from v by a shortest-path of length at most 2^l. Each substree is represented by a multi-set of vertex labels. We build the Weisfeiler-Lehman dictionary by finding all subtree representations of every graph in the dataset. The graph feature or fingerprint is a frequency vector in which each component corresponds to the frequency of a particular dictionary element.

Formally, we have a set of N graphs $\mathcal{G} = \{G^{(1)}, .., G^{(N)}\}$ where $G^{(i)} = (V^{(i)}, E^{(i)})$ $(1 \leq i \leq N)$. Let $f^G : V \rightarrow \Omega$ be the initial feature vector for each vertex of graph $G = (V, E)$. Let $S_k^G(v)$ be the set of vectors for vertex $v \in V$ of graph G at Weisfeiler-Lehman level k.

```
function Dictionary Weisfeiler-Lehman
01.  Universal dictionary: $\mathcal{D} \leftarrow \emptyset$
02.  for $i = 1 \rightarrow N$:
03.    Initialize the WL level 0
04.    for each $v \in V^{(i)}$:
05.      $S_0^{G^{(i)}}(v) \leftarrow \{f^{G^{(i)}}(v)\}$
06.      $\mathcal{D} \leftarrow \mathcal{D} \cup \{S_0^{G^{(i)}}(v)\}$
07.    end for
08.    Build the WL level 1, 2, .., $K$
09.    for $k = 1 \rightarrow K$:
10.      for each $v \in V^{(i)}$:
11.        $S_k^{G^{(i)}}(v) \leftarrow S_{k-1}^{G^{(i)}}(v)$
12.        for each $(u, v) \in E^{(i)}$:
13.          $S_k^{G^{(i)}}(v) \leftarrow S_k^{G^{(i)}}(v) \cup S_{k-1}^{G^{(i)}}(u)$
14.        end for
15.        $\mathcal{D} \leftarrow \mathcal{D} \cup \{S_k^{G^{(i)}}(v)\}$
16.      end for
17.    end for
18.  end for
19.  $\mathcal{S} \leftarrow \{S_0^{G^{(1)}}, .., S_0^{G^{(N)}}, .., S_K^{G^{(1)}}, .., S_K^{G^{(N)}}\}$
20.  return $\mathcal{D}, \mathcal{S}$
end function
```

3.1.3. Histogram-Alignment Graph Feature

Optimal-assignment Weisfeiler-Lehman graph kernel [4] ultilizes the original Weisfeiler-Lehman graph kernel by applying the Hungarian matching algorithm for bipartite graphs to find the optimal matching between two sets of vertex feature vectors of two graphs. One drawback of this approach is the time complexity of the matching algorithm. To address this problem, we construct a multi-level histogram of frequencies as the fingerprint for each graph. This method is known as *Histogram-alignment Weisfeiler-Lehman graph feature*.

3.2. 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 zero-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 3-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.

3.2.1. 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 $f(v) \in \Re^d$ 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 $W_i \in \Re^{d \times d}$ 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 $u \in \Re^d$. 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 $h_t(v) \in \Re^d$ 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:

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

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 $H_t \in \Re^{|V| \times d}$ denote the vertex embedding matrix in which the v-th row of H_t is the embedding of vertex v at iteration t. Equation (1) is equivalent to:

$$H_t = (I_{|V|} + D^{-1} \cdot A) \cdot H_{t-1} \tag{2}$$

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]:

$$H_t = (I_{|V|} - D^{-1/2} A D^{-1/2}) \cdot H_{t-1} \tag{3}$$

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 $W_t \in \Re^{d \times d}$ 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 $\mathcal{L}_G \in \Re$. The forward pass of the graph neural network (GNN) is described by the following algorithm:

```
function Forward($G = (V, E, A), T \in \mathbb{N}$)
01.    Initialize $W_0, W_1, .., W_T \in \Re^{d \times d}$
02.    Layer 0: $L_0 = \sigma(H_0 \cdot W_0)$
03.    Layer $t \in \{1, .., T\}$: $L_t = \sigma(H_t \cdot W_t)$
04.    Compute the graph feature: $f_G = \sum_{v \in V} L_T(v) \in \Re^d$
05.    Linear regression on layer $T + 1$
06.    Minimize: $\|\langle u, f_G \rangle - \mathcal{L}_G\|_2^2$ where $u \in \Re^d$ is learnable
end function
```

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.

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

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

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

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

where $h_{l-1}(u) \otimes h_{l-1}(w) \otimes h_{l-1}(t) \in \Re^{d \times d \times d}$ is the tensor product of 3 rank-1 vectors, and we obtain zero-order $\phi_l^{cubic}(v) \in \Re^d$ by taking the diagonal of the 3-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 3.3. Covariant Compositional Networks, we will discuss further this hierarchical extension of neighborhood and definition of a receptive field.

3.2.2. 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 the following algorithm:

```
function Contruct-Neighbor ($v \in V$)
01.    $\Omega(v) \leftarrow \emptyset$
02.    for each distance $l \in 0, .., |V| - 1$:
03.        for each vertex $w \in V$:
04.            if $d(v, w) = l$:
05.                $\Omega(v) \leftarrow \Omega(v) \cup \{w\}$
06.            end if
07.        end for
08.        if $|\Omega(v)| \geq K$:
09.            break
```

```
10.         end if
11.     end for
12.     if $|\Omega(v)| < K$:
13.         Add $K - |\Omega(v)|$ dummy vertices into $\Omega(v)$
14.     end if
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 $\Omega(v)$
end function
```

We also have the algorithm to flatten the input graph G as follows into a sequence of $N \times K$ vertices:

```
function Flatten-Graph ($G = (V, E)$)
01.     Suppose that $V = \{v_1, .., v_{|V|}\}$
02.     Sort $\bar{V} \leftarrow \{v_{i_1}, .., v_{i_{|V|}}\}$ such that $\sigma(v_{i_t}) < \sigma(v_{i_{t+1}})$
03.     Output sequence $S \leftarrow \emptyset$
04.     for each $v \in \bar{V}$:
05.         Add $\Omega(v)$ at the end of $S$
06.     end for
07.     return $S$
end function
```

Suppose that each vertex is associated with a fixed-size input feature vector of L channels. By the Flatten-Graph algorithm, we can produce a feature matrix of size $L \times (NK)$. We can apply the standard convolutional 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.

3.3. Covariant Compositional Networks

3.3.1. 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 zero-order representation limits $f_l(v)$ to be a vector of d channels as discussed in section 3.2.

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, $f_l(v) \in \Re^{|\Omega_l(v)| \times d}$ where each row of $f_l(v)$ is a zero-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 3-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 $f_{l-1}(w) \in \Re^{|\Omega_{l-1}(w)| \times d}$ 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 $\phi_l^{first}(v) \in \Re^{|\Omega_l(v)| \times d}$. 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 $f_{l-1}(w) \in \Re^{|\Omega_{l-1}(w)| \times |\Omega_{l-1}(w)| \times d}$ for each w, operator $\otimes$ is the broad-casting matrix-tensor multiplication, and $X_l(v, w) \in \Re^{|\Omega_l(v)| \times |\Omega_{l-1}(w)|}$ is the permutation matrix defined as above. The second-order aggregation rule gives us the second-order representation $\phi_l^{second}(v) \in \Re^{|\Omega_l(v)| \times |\Omega_l(v)| \times d}$.

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 zero-order, first-order or second-order aggregation rules; learnable weight matrix $W_l \in \Re^{d \times d}$; learnable bias vector $b_l \in \Re^d$; 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 $W_l \in \Re^{\lfloor d/2 \rfloor \times d}$ and $b_l \in \Re^{\lfloor d/2 \rfloor}$.

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.

3.3.2. Tensor Stacking

To empower the first-order and second-order representations, instead of summing up the lower-level vertex representations as in section 3.3.2. Generic Algorithm, we stack them into a higher-order tensor:

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

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

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

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

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

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

3.3.3. *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 6-order tensor:

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

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.

3.3.4. *Virtual Indexing System*

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

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

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

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.

3.3.5. *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:

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

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

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

Again, we cannot implement the tensor contraction explicitly but via the **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:

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

and the second-order representation is:

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

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

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

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

3.3.6. *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:

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

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.

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:

- 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 $\phi_l(v) \in \Re^{|\Omega_l(v)| \times |\Omega_l(v)| \times |\Omega_l(v)| \times d}$ 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 6-order into 3-order tensor of size $|\Omega_l(v)| \times |\Omega_l(v)| \times d$. As discussed section 3.3.5., 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.

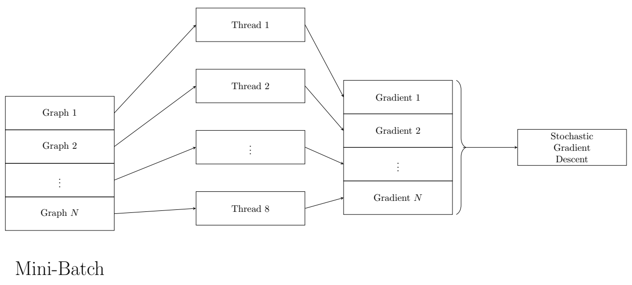
4.3.4. CPU Multi-threading In Gradient Computation

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 the following figure:

Figure 2. CPU multi-threading for gradient computation



5. Experiments and Results

5.1. Matrix Multiplication

To show the efficiency of our GPU matrix multiplication, we establish several performance tests and measure the running time to compare with an $O(N^3)$ CPU matrix multiplication. The sizes are the matrices are $N \in \{128, 256, 512, 1024\}$. In the largest case, we observe that GPU gives a factor of 200x improvement. The following table shows the detail:

Method	N = 128	N = 256	N = 512	N = 1024
CPU	22 ms	379 ms	2,274 ms	15,932 ms
GPU	< 1 ms	4 ms	15 ms	70 ms

Table 1. GPU vs CPU matrix multiplication

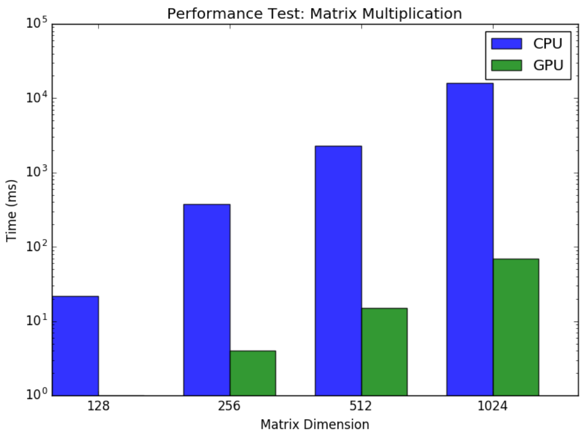


Figure 3. GPU vs CPU matrix multiplication running time (milliseconds) in $\log_{10}$ scale

5.2. Tensor Contraction

We also compare the GPU implementation of tensor contraction with the CPU one. We want to remark that the tensor contraction complexity is $O(18 \times |\Omega_l(v)|^5 \times d)$ that grows exponentially with the size of receptive field $|\Omega_l(v)|$ and grows linearly with the number of channels. We have a constant 18 as the number of unique contractions in the second-order case. We have several tests with the size of receptive field $|\Omega_l(v)|$ ranging in $\{5, 10, 20, 35\}$ and the number of channels d ranging in $\{10, 20\}$. In the largest case with $|\Omega_l(v)| = 35$ and $d = 20$, we observe that GPU gives a factor of approximately 62x speedup. The following table shows the detail:

Figure 4. GPU vs CPU tensor contraction running time (milliseconds) in $\log_{10}$ scale

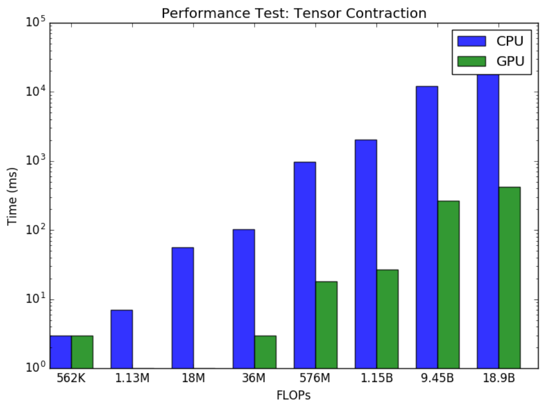


Table 2. GPU and CPU tensor contraction

$\lvert\Omega_l(v)\rvert$	d	Floating-points	CPU	GPU
5	10	562,500	3 ms	3 ms
5	20	1,125,000	7 ms	1 ms
10	10	18,000,000	56 ms	1 ms
10	20	36,000,000	103 ms	3 ms
20	10	576,000,000	977 ms	18 ms
20	20	1,152,000,000	2,048 ms	27 ms
35	10	9,453,937,500	12,153 ms	267 ms
35	20	18,907,875,000	25,949 ms	419 ms

5.3. Putting all operations together

In this experiment, we generate synthetic random input graphs by Erdos-Renyi $p = 0.5$ model. The number of vertices $|V| \in \{10, 15, 20, 25\}$. We fix the maximum size of receptive field $|\Omega_l(v)|$ as 10 and 15, the number of channels d as 10, and the number of levels / layers of the neural network L as 6. In the largest case of the graph with 25 vertices, GPU gives a factor of approximately 6x speedup. The following table shows the detail:

Table 3. GPU and CPU network evaluation

$\lvert V\rvert$	Max $\lvert\Omega_l(v)\rvert$	d	L	CPU	GPU
10	10	10	6	1,560 ms	567 ms
15	10	10	6	1,664 ms	543 ms
20	15	10	6	7,684 ms	1,529 ms
25	15	10	6	11,777 ms	1,939 ms

5.4. Small Molecular Dataset

This is the total training and testing time on a small dataset of 4 molecules CH_4, NH_3, H_20, C_2H_4 with 1,024 epochs. After each epoch, we evaluate the neural network immediately. CCN 1D denotes the Covariant Compositional Networks with the first-order representation, the number of layers / levels is in $\{1, 2, 4, 8, 16\}$. CCN 2D denotes the Covariant Compositional Networks with the second-order representation, the number of layers / levels is in $\{1, 2\}$. The number of channels $d = 10$ in all settings. In this experiment, we use 4 threads for the training minibatches of 4 molecules and compare the running time with the single thread case. All models are fully converged.

Table 4. Single thread vs Multiple threads

Model	Layers	Single-thread	Multi-thread
CCN 1D	1	1,836 ms	874 ms
CCN 1D	2	4,142 ms	1,656 ms
CCN 1D	4	9,574 ms	3,662 ms
CCN 1D	8 (deep)	20,581 ms	7,628 ms
CCN 1D	16 (very deep)	42,532 ms	15,741 ms
CCN 2D	1	35 seconds	10 seconds
CCN 2D	2	161 seconds	49 seconds

5.5. 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 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.

5.5.1. *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.

Figure 5. WL feature vectors with PCA and t-SNE

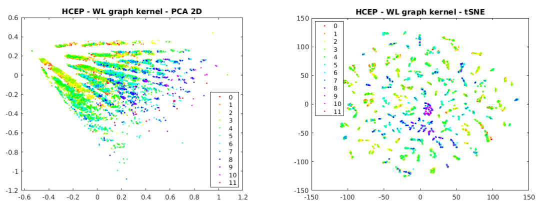


Figure 6. CCN 1D feature vectors with PCA and t-SNE

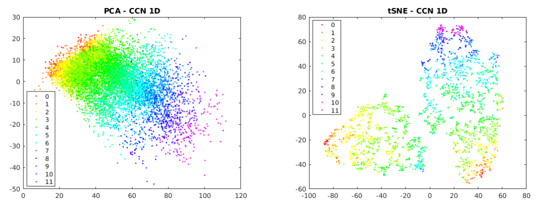
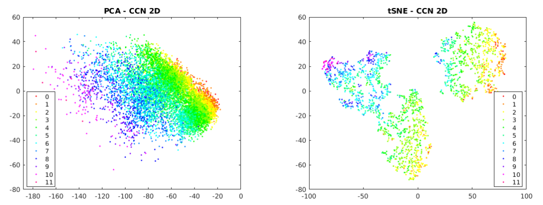


Figure 7. CCN 2D feature vectors with PCA and t-SNE



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.

5.5.2. *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 $C = 100$.

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.

Table 5. HCEP dataset

Model	Test MAE	Test RMSE
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

Figure 8. Testing errors vs Number of training epochs. Top: MAE. Down: RMSE

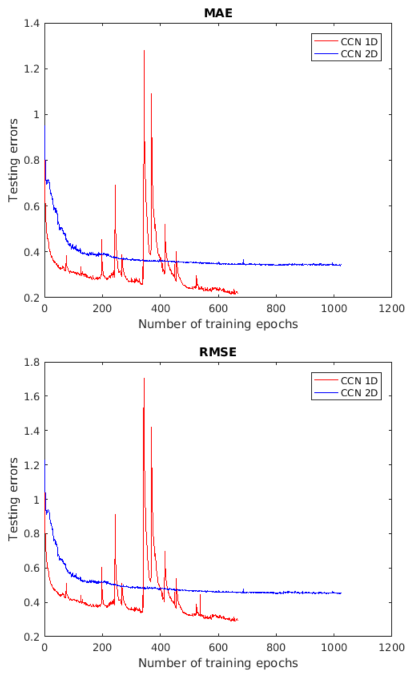
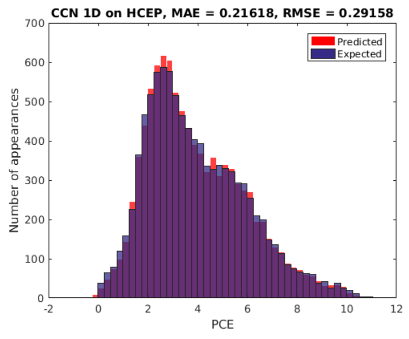
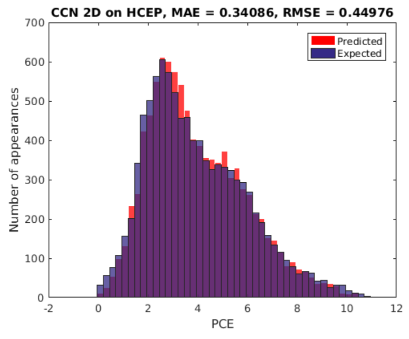


Figure 9. Distributions of the predicted PCE values and the expected ones. Top: CCN 1D. Down: CCN 2D





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 a subset the Harvard Clean Energy Project dataset. 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.

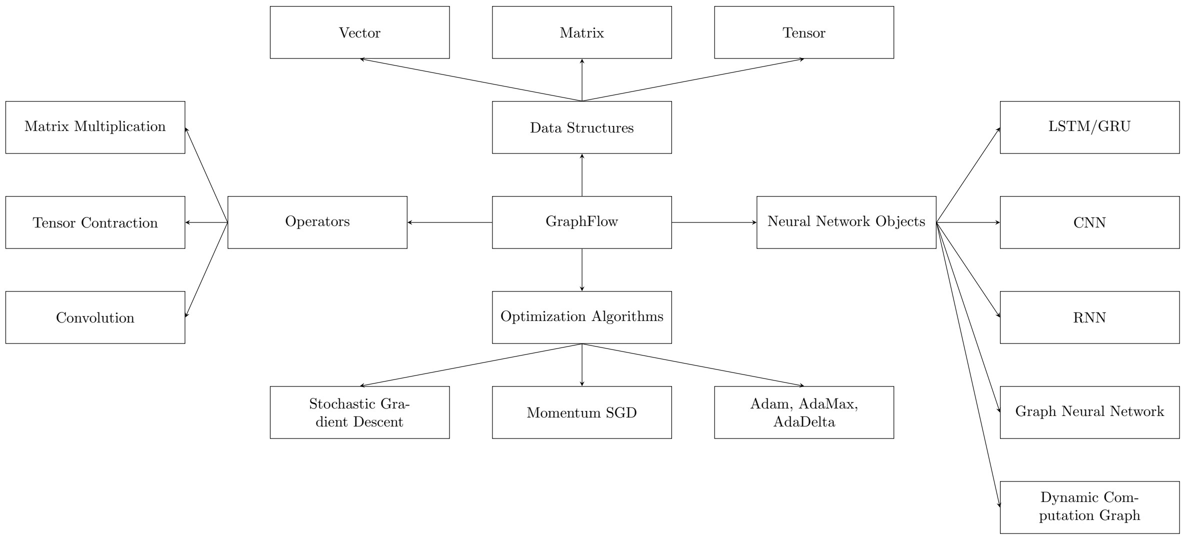
[13] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, “Automatic differentiation in PyTorch”, *Neural Information Processing Systems (NIPS)*, 2017.

[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.

9. Appendix


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