THE UNIVERSITY OF CHICAGO

COVARIANT COMPOSITIONAL NETWORKS FOR LEARNING GRAPHS AND GRAPHFLOW DEEP LEARNING FRAMEWORK IN C++/CUDA

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
HY TRUONG SON

CHICAGO, ILLINOIS
2018

For my father, my mother and all my loved ones.

Victory belongs to the most persevering.

– Napoleon Bonaparte

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Professor Risi Kondor – who has supported me, helped me, guided me, and conveyed continually and convincingly to all of us, his students, a spirit of discovery in regard to research and scholarship. A very special thanks to the entire UChicago Machine Learning group and Department of Computer Science. Finally, I would like to thank Google Inc., especially Networking SRE, Security & Privacy and Brain teams, for their inspiration and the internship opportunities for me to apply my research into practice.

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 GPU. 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 graph learning models in Harvard Clean Energy Project and QM9 molecular datasets.

Index Terms: graph neural network, message passing, density functional theory, deep learning framework

CHAPTER 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 (Shervashidze et al., 2011). However, a limitation of kernel methods (see section 2) 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 2.4 and 2.5. The common idea of family of Weisfeiler-Lehman graph kernel is hashing the substructures 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 sections from 2.9 to 2.19). We detail the graph neural network baselines such as Neural Graph Fingerprint (Duvenaud et al., 2015) and Learning Convolutional Neural Networks (Niepert et al., 2016) in sections 2.10 and 2.11. In the context of graphs, the substructures 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 chapter 3. Also in this chapter, we introduce the notions of tensor contractions and tensor products (see section 3.6) to keep the orders of tensors manageable without exponentially growing. Our generalized convolution graph neural network is named as Covariant Compositional Networks (CCNs) (Kondor et al., 2018; Hy et al., 2018). We propose 2 specific algorithms that are first-order CCN in section 3.3 and second-order CCN in section 3.4. 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 3.8).

Current Deep Learning frameworks including TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2017), Mxnet (Chen et al., 2016), Theano (Al-Rfou et al., 2016), 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. Finally, we apply our methods to the Harvard Clean Energy Project (HCEP) (Hachmann et al., 2011) and QM9 (Ramakrishnan et al., 2014) molecular dataset. The visualizations, experiments and empirical results are detailed in chapter 5. Section 6 is our conclusion and future research direction.

CHAPTER 2

GRAPH KERNELS AND GRAPH NEURAL NETWORKS

2.1 Definition Of Graph Kernel

Kernel-based algorithms, such as Gaussian processes (Mackay, 1997), support vector machines (Burges, 1998), and kernel PCA (Mika et al., 1998), 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 points in $\mathcal{X}$ in terms of a kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ (Hofmann et al., 2008). The kernel function $k(\cdot, \cdot)$ is required to satisfy that for all $x, x' \in \mathcal{X}$,

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle \tag{2.1}$$

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

$$K \triangleq (k(x_i, x_j))_{ij} \tag{2.2}$$

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

$$c^T K c = \sum_{i,j} c_i c_j K_{ij} \geq 0 \tag{2.3}$$

for all $c \in \mathbb{R}^n$ is called positive definite. If equality in 2.3 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

$$\sum_{i,j} c_i c_j k(x_i, x_j) \geq 0 \tag{2.4}$$

holds for any $n \in \mathbb{N}$, $\{x_1, .., x_n\} \subseteq \mathcal{X}^n$ and $c \in \mathbb{R}^n$. The inequality 2.4 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:

$$\mathcal{K}_{graph} \triangleq \frac{1}{|V_1|} \cdot \frac{1}{|V_2|} \cdot \sum_{v_1 \in V_1} \sum_{v_2 \in V_2} k_{base}(f(v_1), f(v_2)) \tag{2.5}$$

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

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

$$\forall (u, v) \in E_1 : (\sigma(u), \sigma(v)) \in E_2 \tag{2.6}$$

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

$$\forall v \in V_1 : l(v) = l(\sigma(v)) \tag{2.7}$$

The algorithm of Weisfeiler-Lehman (WL) graph isomorphism test (Weisfeiler & Lehman, 1968) is described as follows. We can see that if G_1 and G_2 are isomorphic then the WL test

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 $i \in [0, T]$.

```
for i = 0 → T do
    Compute the multiset of labels M_i(v), string s_i(v) and compressed label f_i(v)
    for v ∈ V do
        if i = 0 then
            M_i(v) ← ∅
            s_i(v) ← l(v)
            f_i(v) ← h(s_i(v))
        else
            M_i(v) ← {f_{i−1}(u)|u ∈ N(v)} where N(v) = {u|(u, v) ∈ E}
            Sort M_i(v) in ascending order and concatenate all labels of M_i(v) into
             string s_i(v)
            s_i(v) ← s_i(v) ⊕ f_{i−1}(v) where ⊕ is concatenation operation.
            f_i(v) ← h(s_i(v))
        end
    end
end
```

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 (Babai & Kucera, 1979). Suppose that we have an efficient

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.

```
Apply algorithm 1 on graph G_1 to get {f_i^{G_1}(v)}
Apply algorithm 1 on graph G_2 to get {f_i^{G_2}(v)}
for i = 0 → T do
    F_i^{G_1} ← {f_i^{G_1}(v) | v ∈ V_1}
    F_i^{G_2} ← {f_i^{G_2}(v) | v ∈ V_2}
    Sort F_i^{G_1} in ascending order
    Sort F_i^{G_2} in ascending order
    if F_i^{G_1} ≢ F_i^{G_2} then
        return G_1 and G_2 are not isomorphic
    end
end
return G_1 are G_2 are isomorphic
```

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 $|E| = |E_1| + |E_2|$.

2.3 Weisfeiler-Lehman Graph Kernel

Based on algorithms 1 2 and equation 2.5, 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:

$$l(v) \triangleq |\{u|(u,v) \in E\}| = |\mathcal{N}(v)| \tag{2.8}$$

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 (Shervashidze et al., 2011)

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.

Apply algorithm 1 on graph G_1 to get $\{f_i^{G_1}(v)\}$

Apply algorithm 1 on graph G_2 to get $\{f_i^{G_2}(v)\}$

for $v \in V_1$ **do**

  $f^{G_1}(v) \leftarrow \bigoplus_{i=0}^{T} f_i^{G_1}(v)$

end

for $v \in V_2$ **do**

  $f^{G_2}(v) \leftarrow \bigoplus_{i=0}^{T} f_i^{G_2}(v)$

end

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

return $\mathcal{K}_{graph}(G_1, G_2)$

2.4 Dictionary WL Graph Feature

We combine the Weisfeiler-Lehman graph kernel and Morgan circular fingerprints into the Dictionary Weisfeiler-Lehman graph feature algorithm. To capture the local substructures of a graph, we define a Weisfeiler-Lehman substree at level/iteration n-th rooted at a vertex v to be the shortest-path subtree that includes all vertices reachable from v by a path of length at most 2^n. Each subtree is represented by a multiset of vertex labels. We build

the Weisfeiler-Lehman dictionary by finding all subtree representations of every graph in the dataset (as in algorithm 6). The graph feature or fingerprint is a frequency vector in which each component corresponds to the frequency of a particular dictionary element (as in algorithm 7).

The remaining question is: How to construct the subtree representation properly? We propose the following solution. Given a multiset of vertex labels M of a subtree, we sort all vertex labels of M in ascending lexicographic order, and concatenate all of them into a string $s(M)$. Finally, we update the dictionary with an element $s(M)$.

One problem of this approach is: Two subtrees with different structures can have the same representing multiset of vertex labels, and is represented by the same element in the dictionary. Definitely we need a more sophisticated representation that is permutation-invariant with respect to the ordering of vertices, while being able to capture the local structures of a graph. This will be fully addressed and discussed in the next chapter with our proposed model Covariant Compositional Networks.

Regarding data structures to implement our algorithms, we need to have an efficient algorithm for the insertion and searching operations with the dictionary D that can contain millions strings in practice. Our choice is Trie data structure or in another name as Prefix tree. Trie's idea was first introduced by (Briandais, 1959) and then by (Fredkin, 1960). Suppose that our strings only contain ASCII characters indexed from 0 to 255. Each node of the Trie/Prefix tree has exactly 256 pointers, we organize these pointers in an array `next`, each element of `next` corresponds to a character in the ASCII table. There is only a single root node in the Trie/Prefix tree. At the beginning, the Trie only contains the root node with 256 pointers pointing to `NULL`. The insertion operation of a string s to the Trie D can

be described as follows. We have a pointer p pointing to the root node at first. We go from left to right in the string s. For each character `s[i]`, if `p -> next[s[i]] = NULL` then we allocate a new memory location for `p -> next[s[i]]`, and move `p` to `p -> next[s[i]]`. In the end of the insertion algorithm, we mark `p` as having a string ended at itself or we increase the number of strings ended there. The searching operation can be efficiently implemented in a similar way, except that we do not create any new memory location, but stop searching when we encounter `NULL`.

Suppose that we can access any element of the array of pointers `next` in a constant time $O(1)$.

Algorithm 4: Insertion into Trie

Data: String s and Trie D passed by reference with the `root` node
Result: Updated Trie D

```
1 Assign p to root
2 for i = 1, 2, .., |s| do
3 |   Allocate a new memory location for p -> next[s[i]] if it is NULL
4 |   Assign p to p -> next[s[i]]
5 end
6 Mark p as having a string ended at p, or increase the number of strings ended at p.
```

Algorithm 5: Searching in Trie

Data: String s and Trie D passed by reference with the `root` node
Result: Return `true` if found, otherwise return `false`

```
1 Assign p to root
2 for i = 1, 2, .., |s| do
3 |   Return false immediately if p -> next[s[i]] = NULL
4 |   Assign p to p -> next[s[i]]
5 end
6 Return false if there is no string ended at p, otherwise return true.
```

The time complexity of both insertion and searching operations of string s in Trie/Prefix tree is $O(|s|)$ where $|s|$ is length of the string s.

Remark that statements $M \leftarrow M \cup s(S_l(v))$ and $D \leftarrow D \cup s(S_l(v))$ in algorithm 6 will

Algorithm 6: Finding all dictionary elements representing a graph

Data: Given a graph $G = (V, E)$, label $l(v)$ associated with each vertex $v \in V$, number of iterations T, and the dictionary D passed by reference. Let $S_l(v)$ denote the multiset of labels of the subtree rooted at v in level l-th. Suppose $s(S)$ is a function returns an unique string representing for the multiset of labels S.

Result: Updated dictionary D with new elements found from G, and M is a multiset containing dictionary elements representing G.

```
Initialize the WL level 0:
for v ∈ V do
    S_0(v) ← {l(v)}
end
Build the WL level 1, 2, .., T:
for l = 1 → T do
    for v ∈ V do
        S_l(v) ← S_{l−1}(v)
        for (u, v) ∈ E do
            S_l(v) ← S_l(v) ∪ S_{l−1}(u)
        end
    end
end
Update dictionary D and build the multiset of dictionary elements M:
M ← ∅
for l = 0 → T do
    for v ∈ V do
        M ← M ∪ s(S_l(v))
        D ← D ∪ s(S_l(v))
    end
end
return D, M
```

be implemented efficiently with Trie data structure, the time complexity of both statements is $O(|s(S_l(v))|)$. For the sake of simplicity of the complexity analysis, we assume that all set operations (including operations with Trie) can be done in a constant time. The total complexity of both algorithms 6 and 7 is $O(N \cdot T \cdot |E|)$.

Algorithm 7: Building the dictionary for a dataset of graphs

Data: Given a dataset of N graphs $\mathcal{G} = \{G^{(1)}, .., G^{(N)}\}$ where $G^{(i)} = (V^{(i)}, E^{(i)})$. Let $l_G : V \to \Omega$ be the initial feature vector for each vertex of graph $G = (V, E)$. Ω is the set of all possible vertex labels.

Result: Return the dictionary D and the frequency vector $F^{(i)}$ for each graph $G^{(i)}$. Remark that D is a set of strings, not a multiset. Remark that D is a set of strings, while M is a multiset of strings.

```
Build dictionary D:
D ← ∅
for i = 1 → N do
    Apply algorithm 6 on graph G^(i) to update D and get M^(i)
end
Given the dictionary D, build the frequency vectors:
for i = 1 → N do
    F^(i) ← 0^|D|
    for m ∈ M^(i) do
        Search for m in D, let say m = D_j
        F^(i)_j ← F^(i)_j + 1
    end
    F^(i) ← F^(i)/‖F^(i)‖ where ‖.‖ denotes the norm l_2 of the vector
end
return D, {F^(1), .., F^(N)}
```

2.5 Optimal Assignment Kernel And Histogram-Alignment WL Graph Feature

We define the *optimal assignment kernel* (Kriege et al., 2016) as follows. Let $[\mathcal{X}]^n$ denote the set of all n-element subsets of a set $\mathcal{X}$ and $\mathcal{B}(X, Y)$ denote the set of all bijections between

X, Y in $[\mathcal{X}]^n$ for $n \in \mathbb{N}$. The optimal assignment kernel $K^k_{\mathcal{B}}$ on $[\mathcal{X}]^n$ is defined as

$$K^k_{\mathcal{B}}(X, Y) \triangleq \max_{B \in \mathcal{B}(X,Y)} W(B)$$

where

$$W(B) \triangleq \sum_{(x,y) \in B} k(x, y)$$

and k is a base kernel on $\mathcal{X}$. In order to apply the kernel to sets of different cardinality, i.e. $|X| \neq |Y|$, we fill up the smaller set by additional objects z such that $k(x, z) = 0$ for all $x \in \mathcal{X}$. Finding the optimal B can be formularized as the Hungarian matching problem that can be solved efficiently by Kuhn-Munkres algorithm (Munkres, 1957). As following, we define the strong kernel and state its relation to the validity of an optimal assignment kernel.

Definition 2.5.1 (Strong kernel)**.** A function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ is called **strong kernel** if $k(x, y) \geq \min\{k(x, z), k(z, y)\}$ for all $x, y, z \in \mathcal{X}$.

Theorem 2.5.1 (Validity of an optimal assignment kernel)**.** *If the base kernel k is strong, then the function $K^k_{\mathcal{B}}$ is a valid kernel.*

Proof. Detail of the proof is contained in (Kearns et al., 2016). □

Inspired by the histogram intersection kernel that yields an optimal assignment kernel (Kriege et al., 2016), we introduce the histogram alignment WL graph feature as follows. Suppose that the vertex labels can be discretized and encoded as one-hot vectors of size c. Let $\ell_v \in \{0, 1\}^c$ be the label vector of vertex v. For each vertex v, we consider the histogram of vertex labels of vertices at distance n from v as:

$$h^n_v = \sum_{w \in V : d(v,w) = n} \ell_w$$

where $d(v, w)$ denotes length of the shortest path between v and w. Given depth N, the histogram alignment WL graph feature of vertex v is computed as concatenating all h_v^n:

$$h_v = \bigoplus_{n \in \{0,..,N\}} h_v^n$$

This graph synthesized feature plays an important role in improving the performance of our graph learning algorithms including Covariant Compositional Networks that will be defined in the next chapter. We can also concatenate the dictionary WL and histogram alignment WL graph features into a richer representation.

2.6 Shortest Path Graph Kernel

Before defining the shortest path kernel, we start with all-paths kernel suggested by (Borgwardt & Kriegel, 2005).

Definition 2.6.1 (All-paths kernel)**.** Given two graphs G_1 and G_2. Let $P(G_i)$ be the set of all paths in graph G_i where $i \in \{1, 2\}$. Let k_{path} be a positive kernel on two graphs, defined as the product of kernels on edges and nodes along the paths. We then define an all-paths kernel $k_{all\ paths}$ as

$$k_{all\ paths}(G_1, G_2) \triangleq \sum_{p_1 \in P(G_1)} \sum_{p_2 \in P(G_2)} k_{path}(p_1, p_2)$$

In other words, we define the all-paths kernel as the sum over all kernels on pairs of paths from G_1 and G_2.

Lemma 2.6.1. *The all-paths kernel is positive definite.*

Proof. Detail of the proof is contained in (Borgwardt & Kriegel, 2005). □

Lemma 2.6.2. *Computing the all-paths kernel is NP-hard.*

Proof. Suppose that determining the set of all paths $P(G)$ in a graph $G = (V, E)$ is not NP-hard. There exists a polynomial time (of $|V|$) algorithm to determine whether G has a Hamilton path by checking if $P(G)$ contains a path of length $|V| - 1$. However, this problem is known to be NP-complete. Therefore, determining the set of all paths and computing the all-paths kernel are NP-hard problems. □

From Lemma 2.6.2, we conclude that the all-paths kernel is infeasible for computation. Thus, as following, we consider the shortest-path kernel. First, for the sake of mathematical convenience to define the shortest-path kernel, we introduce the Floyd-transformed graph.

Definition 2.6.2. Given an undirected connected weighted graph $G = (V, E)$, the Floyd-transformed graph of G is a complete graph $S = (V, \overline{E})$ in which for all $(u, v) \in \overline{E}$, the weight of edge (u, v) is length of the shortest-path between u and v in G.

We can easily construct the Floyd-transformed graph S of G by Floyd-Warshall algorithm in $O(|V|^3)$. After Floyd-transformation of the input graphs, a definition of shortest-path kernel is introduced as follows.

Definition 2.6.3 (Shortest-path graph kernel)**.** Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two graphs that are Floyd-transformed into $S_1 = (V_1, \overline{E}_1)$ and $S_2 = (V_2, \overline{E}_2)$. The shortest-path graph kernel is defined as

$$k_{shortest\ paths}(G_1, G_2) \triangleq \sum_{e_1 \in \overline{E}_1} \sum_{e_2 \in \overline{E}_2} k^{(1)}_{walk}(e_1, e_2)$$

where $k^{(1)}_{walk}$ is a positive definite kernel on edge walks of length 1.

Label enrichment can also be applied to Floyd transformed graphs to speedup kernel computation. When performing the Floyd-Warshall algorithm, besides storing the shortest path information for each pair of vertices, we also store the number of edges on the shortest path. The equal length shortest-path kernel can be done by setting kernels to zero for all

pairs of shortest paths where the number of edges in the shortest paths is not identical (Borgwardt & Kriegel, 2005). Regarding the dynamic programming (DP) algorithm, let $d_{u,v}$ and $d_{u,v}^{(e)}$ denote lengths of the shortest path and the shortest path with exactly e edges between $u, v \in V$. The DP formula of Floyd-Warshall is

$$d_{u,v} = \min_{k \in V} \{d_{u,k} + d_{k,v}\}$$

while the DP formula of Floyd-Warshall with number of edges information is

$$d_{u,v}^{(e)} = \min_{k \in V, e_1 + e_2 = e} \{d_{u,k}^{(e_1)} + d_{k,v}^{(e_2)}\}$$

For a given e, we construct the corresponding Floyd transformed graph of shortest paths with length exactly e edges. From here, we have our definition of Equal-length Shortest-path (ELSP) kernel.

Definition 2.6.4 (Equal-length Shortest-path kernel)**.** Given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. For a non-zero natural number $e \in [1, \min(|V_1|, |V_2|) - 1]$, we construct the Floyd transformed graphs of e-edge shortest paths $S_1 = (V_1, E_1^{(e)})$ and $S_2 = (V_2, E_2^{(e)})$ for G_1 and G_2, respectively. The equal-length shortest-path kernel is defined as

$$k_{ELSP}(G_1, G_2) \triangleq \sum_{e=1}^{\min(|V_1|,|V_2|)-1} \sum_{e_1 \in E_1^{(e)}} \sum_{e_2 \in E_2^{(e)}} k_{walk}^{(1)}(e_1, e_2)$$

2.7 Graphlet Kernel

State-of-the-art graph kernels do not scale to large graphs with hundreds of vertices and thousands of edges. To address this issue, (Shervashidze et al., 2009) proposed Graphlet kernel exploiting frequent subgraph mining algorithms that aims to detect subgraphs that are frequent in a given dataset of graphs. Two graphs $G = (V, E)$ and $G' = (V', E')$ are

isomorphic (denoted by $G \cong G'$) if there exists a bijective mapping $f : V \to V'$ (called the isomorphism function) such that $(v_i, v_j) \in E$ if and only if $(f(v_i), f(v_j)) \in E'$. In graph theory, graphlet is a small, connected, non-isomorphic, induced subgraph (that must contain all edges between its vertices) of a given graph. Let $\mathcal{G}_n = \{g(1), .., g(N_n)\}$ be the set of size-n graphlets. It is trivial to see that:

- $n = 1$: $N_1 = 1$ and $\mathcal{G}_1$ contains only 1 graph that has 1 vertex.
- $n = 2$: $N_2 = 2$ and $\mathcal{G}_2$ contains 2 graphs of 2 vertices, one graph has no edge, and the another one has 1 edge.
- $n = 3$: $N_3 = 4$, there are 4 non-isomorphic graphs with 3 vertices.
- $n = 4$: $N_4 = 11$, there are 11 non-isomorphic graphs with 4 vertices.

Given a graph G, define a vector $f_G^{(n)}$ of length N_n whose i-th component corresponds to the frequency of occurrence of $g(i)$ in G. We will call $f_G^{(n)}$ the n-spectrum of G. This statistic is the foundation of our novel graph kernel.

Definition 2.7.1 (Graphlet kernel)**.** Given two graphs G and G' of size greater than n, the graphlet kernel of graphlet size n is defined as:

$$k_{graphlet}^{(n)}(G, G') \triangleq \langle f_G^{(n)}, f_{G'}^{(n)} \rangle$$

In order to account for differences in the sizes of the graphs, we normalize the frequency counts $f_G^{(n)}$ to probability vectors:

$$k_{graphlet}^{(n)}(G, G') \triangleq \frac{\langle f_G^{(n)}, f_{G'}^{(n)} \rangle}{\left\| f_G^{(n)} \right\|_1 \cdot \left\| f_{G'}^{(n)} \right\|_1}$$

In the precomputation step, we need to find the set $\mathcal{G}_n$ of N_n non-isomorphic graphs of size n. There are $\binom{|V|}{n}$ size-n subgraphs in a graph G, computing $f_G^{(n)}$ requires $O(|V|^n)$.

For each subgraph of G, we need to classify it into 1 element of $\mathcal{G}_n$ by a graph isomorphism test that is well-known to be a NP-complete problem. Therefore, the computation time for graphlet kernel with large n is still extremely expensive.

2.8 Random Walk Graph Kernel

Generalized random walk graph kernels are based on a simple idea: given a pair of graphs, perform random walks on both, and count the number of matching walks (Vishwanathan et al., 2010). First, we construct the direct product graph of two graphs.

Definition 2.8.1 (Kronecker product)**.** Given real matrices $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{p \times q}$, the Kronecker product $A \otimes B \in \mathbb{R}^{np \times mq}$ is defined as:

$$A \otimes B \triangleq \begin{bmatrix} A_{11}B & A_{12}B & \dots & A_{1m}B \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1}B & A_{n2}B & \dots & A_{nm}B \end{bmatrix}$$

Definition 2.8.2 (Direct product graph)**.** Given two graphs $G = (V, E)$ and $G' = (V', E')$ with $|V| = n$ and $|V'| = n'$, their direct product $G_\times$ is a graph with vertex set

$$V_\times = \{(v_i, v'_{i'}) : v_i \in V, v'_{i'} \in V'\}$$

and edge set

$$E_\times = \{((v_i, v'_{i'}), (v_j, v'_{j'})) : (v_i, v_j) \in E \wedge (v'_{i'}, v'_{j'}) \in E'\}$$

If A and A' are the respective adjacency matrices of G and G', then the adjacency matrix of $G_\times$ is $A_\times = A \otimes A'$.

In other words, $G_\times$ is a graph over pairs of vertices from G and G', and two vertices in $G_\times$ are neighbors if and only if the corresponding vertices in G and G' are both neighbors.

Performing a random walk on the direct product graph is equivalent to performing a simultaneous random walk on G and G'. Let p and p' denote the starting probability distributions over the vertices of G and G'. Let q and q' denote the stopping probability distributions over the vertices of G and G'. The corresponding starting and stopping probabilities on the direct product graph are $p_\times = p \otimes p'$ and $q_\times = q \otimes q'$, respectively. Let $|V| = n$ and $|V'| = n'$. If G and G' are edge-labeled (discrete labels), we can associate a weight matrix $W_\times \in \mathbb{R}^{nn' \times nn'}$ with $G_\times$:

$$W_\times = \sum_{\ell=1}^{d} A^\ell \otimes A'^\ell \tag{2.9}$$

where d is the number of different labels, A^ℓ and A'^ℓ are the filtered adjacency matrices of G and G' by label ℓ (we keep the edge weights of edges with label ℓ, and set edge-weight 0 to all other edges). In the case the graphs are unlabeled, we can set $W_\times = A_\times$. Let $A_\times^k$ be the probability of simultaneous length k random walks on G and G'. Let $W_\times^k$ be the similarity between simultaneous length k random walks on G and G', measured via a kernel function $\mathcal{K}$. Given initial and stopping probability distributions $p_\times$ and $q_\times$ one can compute $q_\times^T W_\times^k p_\times$, which is expected similarity between simultaneous length k random walks on G and G'.

To define a kernel which computes the similarity between G and G', one natural idea is to simply sum up $q_\times^T W_\times^k p_\times$ for all values of k. To achieve the convergence, we introduce appropriate chosen non-negative coefficients $\mu(k)$ in the definition of kernel between G and G':

$$k(G, G') \triangleq \sum_{k=0}^{\infty} \mu(k) q_\times^T W_\times^k p_\times \tag{2.10}$$

Based on the generic equation 2.10 of random walk kernel, several authors have defined special cases in the literature.

Definition 2.8.3 (Special case 1 - (Kashima et al., 2004))**.** Assume that $\mu(k) = \lambda^k$ for some

$\lambda > 0$, we can write:

$$k(G, G') \triangleq \sum_{k=0}^{\infty} \lambda^k q_\times^k W_\times^k p_\times = q_\times^T (I - \lambda W_\times)^{-1} p_\times$$

Definition 2.8.4 (Special case 2 - (Gärtner et al., 2003))**.** Assuming uniform distributions for the starting and stopping probabilities over the vertices of G and G', the kernel can be defined as counting the number of matching walks:

$$k(G, G') \triangleq \frac{1}{n^2 n'^2} \sum_{i=1}^{n} \sum_{j=1}^{n'} \sum_{k=0}^{\infty} \mu(k) [A_\times^k]_{ij}$$

Definition 2.8.5 (Special case 3 - (Vishwanathan, 2002))**.** *Exponential* random walk kernel can be defined as:

$$k(G, G') \triangleq \sum_{i=1}^{n} \sum_{j=1}^{n'} [e^{\lambda A_\times}]_{ij}$$

2.9 Message Passing Framework

2.9.1 Label Propagation Algorithm

Label propagation algorithm, or in general the Message Passing framework, has been widely applied to various network problems ranging from PageRank for Google search engine to learning representations for molecules with graph neural networks. The core idea can be simply explained in words as follows. Given an input graph / network $G = (V, E)$. Initially, each vertex v of the graph is associated with a feature representation l_v (label) or f_v^0 (e.g., scalar in PageRank, vector in majority of graph neural networks, or a higher-order tensor in Covariant Compositional Networks). This feature representation can also be called as a *message*. Iteratively, at iteration ℓ, each vertex collects / aggregates all messages of the previous iteration $\{f_{v_1}^{\ell-1}, .., f_{v_k}^{\ell-1}\}$ from other vertices in its neighborhood $\mathcal{N}(v) = \{v_1, .., v_k\}$, and then produces a new message f_v^ℓ via some *hashing function* $\Phi(.)$. The graph representa-

tion $\phi(G)$ is obtained by aggregating all messages in the last iteration of every vertex. The generic algorithm is described in pseudocode 8.

Algorithm 8: Label Propagation Algorithm

for $v \in V$ **do**
  $f_v^0 \leftarrow l_v$
end
for $\ell = 1 \rightarrow L$ **do**
  for $v \in V$ **do**
    $f_v^\ell \leftarrow \Phi(f_{v_1}^{\ell-1}, .., f_{v_k}^{\ell-1})$ where $\mathcal{N}(v) = \{v_1, .., v_k\}$
  end
end
$\phi(G) \leftarrow \Phi(f_1^L, .., f_{|V|}^L)$
Use $\phi(G)$ for downstream regression / classification tasks.

2.9.2 Generic Graph Neural Network

Graph neural networks can be built based on the Message Passing framework in which the hashing function $\Phi(.)$ at iteration ℓ has n_ℓ learnable parameters $\{W_1^\ell, .., W_{n_\ell}^\ell\}$. The gradients of the loss function with respect to these learnable parameters can be computed by Back-Propagation algorithm, similarly to Recurrent Neural Networks. The learnable parameters will then be optimized by Stochastic Gradient Descent (SGD) or its variants. It is required that $\Phi(.)$ must be differentiable. The generic algorithm of a graph neural network is described in pseudocode 9. In later sections of this chapter, we will go through a brief summary for the field of graph neural networks with state-of-the-art algorithms including Neural Graph Fingerprint (see section 2.10), Convolutional Neural Networks for graphs (see section 2.11), Gated Graph Neural Networks (see section 2.12), Weisfeiler-Lehman Networks (see section 2.13), Message Passing Neural Networks (see section 2.14), Interaction Networks (see section 2.15), Molecular Graph Convolutions (see section 2.16), Deep Tensor Neural Networks (see section 2.17), Laplacian Based Methods (see section 2.18), and Graph-based Semi-Supervised Learning (see section 2.19).

Algorithm 9: Generic Graph Neural Network

Initialize learnable parameters $\{W_1^\ell, .., W_{n_\ell}^\ell\}$ for each layer ℓ.
Initialize learnable parameters $\{W_1, .., W_n\}$ for learning the graph representation.
for $v \in V$ **do**
  $f_v^0 \leftarrow l_v$
end
for $\ell = 1 \rightarrow L$ **do**
  for $v \in V$ **do**
    $f_v^\ell \leftarrow \Phi(f_{v_1}^{\ell-1}, .., f_{v_k}^{\ell-1}; \{W_1^\ell, .., W_{n_\ell}^\ell\})$ where $\mathcal{N}(v) = \{v_1, .., v_k\}$
  end
end
$\phi(G) \leftarrow \Phi(f_1^L, .., f_{|V|}^L; \{W_1, .., W_n\})$
Use $\phi(G)$ for downstream regression / classification tasks.

2.10 Neural Graph Fingerprint

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 $\{W_1, .., W_L, u\}$ 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 $W_\ell \in \mathbb{R}^{d \times d}$ associating with level ℓ-th of the neural network. For L levels, we update the vector stored at vertex v using W_ℓ.

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 L-iteration label propagation algorithm on graph G. Let $h_\ell(v) \in \mathbb{R}^d$ be the vertex embedding of vertex v at iteration $\ell \in \{0, .., L\}$. At $\ell = 0$, we initialize $h_0(v) = f(v)$. At $\ell \in \{1, .., L\}$, we update $h_{\ell-1}$ to h_ℓ at a vertex v using the values on v's

neighbors:

$$h_\ell(v) = h_{\ell-1}(v) + \frac{1}{|\mathcal{N}(v)|} \sum_{w \in \mathcal{N}(v)} h_{\ell-1}(w) \tag{2.11}$$

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 $H_\ell \in \mathbb{R}^{|V| \times d}$ denote the vertex embedding matrix in which the v-th row of H_ℓ is the embedding of vertex v at iteration ℓ. Equation 2.11 is equivalent to:

$$H_\ell = (I_{|V|} + D^{-1} \cdot A) \cdot H_{\ell-1} \tag{2.12}$$

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 (Kipf & Welling, 2017):

$$H_\ell = (I_{|V|} - D^{-1/2} A D^{-1/2}) \cdot H_{\ell-1} \tag{2.13}$$

From the label propagation algorithms, we build the simplest form of graph neural networks Kearns et al. (2016); Duvenaud et al. (2015); Kipf & Welling (2017). Suppose that iteration ℓ is associated with a learnable matrix $W_\ell \in \mathbb{R}^{d \times d}$ 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 $\mathcal{T}_G \in \mathbb{R}$. The forward pass of the graph neural network (GNN) is described by algorithm 10. Learnable matrices W_ℓ 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 aggrega-

Algorithm 10: 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 $L \in \mathbb{N}$.

Result: Construct the corresponding neural network.

1 Initialize $W_0, W_1, .., W_L \in \mathbb{R}^{d\times d}$

2 Layer 0: $\mathcal{L}_0 = \sigma(H_0 \cdot W_0)$

3 Layer $\ell \in \{1, .., L\}$: $\mathcal{L}_\ell = \sigma(H_\ell \cdot W_\ell)$

4 Compute the graph feature: $f_G = \sum_{v\in V} \mathcal{L}_L(v) \in \mathbb{R}^d$

5 Linear regression on layer $L + 1$

6 Minimize: $\|\langle u, f_G\rangle - \mathcal{T}_G\|_2^2$ where $u \in \mathbb{R}^d$ is learnable

tion 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 $\ell - 1$ to get a permutation invariant representation of vertex v at level ℓ:

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

where $\phi_\ell^{linear}(v) \in \mathbb{R}^d$ and $h_{\ell-1}(w) \in \mathbb{R}^d$ 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 $\phi_\ell^{quadratic}(v)$:

$$\phi_\ell^{quadratic}(v) = diag\Big(\sum_{u\in\mathcal{N}(v)}\sum_{w\in\mathcal{N}(v)} h_{\ell-1}(u)h_{\ell-1}(w)^T\Big)$$

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

$$\phi_\ell^{cubic}(v) = diag\Big(\sum_{u,w,t\in\mathcal{N}(v)} h_{\ell-1}(u) \otimes h_{\ell-1}(w) \otimes h_{\ell-1}(t)\Big)$$

where $h_{\ell-1}(u) \otimes h_{\ell-1}(w) \otimes h_{\ell-1}(t) \in \mathbb{R}^{d \times d \times d}$ is the tensor product of 3 rank-1 vectors, and we obtain zeroth-order $\phi_{\ell}^{cubic}(v) \in \mathbb{R}^d$ 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 $\ell = 0$: $\mathcal{N}_0(v) = \{v\}$
- At level $\ell > 0$:

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

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

From equation 2.12, we extend the basic GNN as follows. Let $\overline{A}$ be the normalized adjacency matrix (or probability transition matrix) in which $\overline{A}_{ij}$ corresponds to the transition probability from vertex i to vertex j (via only 1 edge). Similarly, $[\overline{A}^k]_{ij}$ corresponds to the probability of a random walk starting at vertex i and ending at vertex j after k edges. Using $\overline{A}^k$ means for each vertex, we consider its neighborhood of distance k by a walk. We have the extension of 2.12:

$$H_{\ell} = \left(\sum_{k=0}^{n} \overline{A}^k \right) \cdot H_{\ell-1} \tag{2.14}$$

where $\overline{A}^0 \equiv I_{|V|}$ and n is the maximum distance for the neighborhood.

2.11 Learning Convolutional Neural Networks For Graphs

The idea of Learning Convolutional Neural Networks for Graphs (LCNN) from (Niepert et al., 2016) 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 $\sigma : V \to \mathbb{R}$ 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 11. We

Algorithm 11: 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 $\Omega(v)$.

$\Omega(v) \leftarrow \emptyset$
for $l \in 0, .., |V| - 1$ **do**
  for $w \in V$ **do**
    if $d(v, w) = l$ **then**
      $\Omega(v) \leftarrow \Omega(v) \cup \{w\}$
    end
  end
  if $|\Omega(v)| \geq K$ **then**
    break
  end
end
if $|\Omega(v)| < K$ **then**
  Add $K - |\Omega(v)|$ dummy vertices into $\Omega(v)$
end
Suppose $\Omega(v) = \{v_1, .., v_K\}$
Sort $\Omega(v) \leftarrow \{v_{i_1}, .., v_{i_K}\}$ such that $\sigma(v_{i_t}) < \sigma(v_{i_{t+1}})$
return $\Omega(v)$

also have algorithm 12 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 12, 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.

Algorithm 12: Flattening the graph

Data: Given an undirected graph $G = (V, E)$.
Result: Sequence S of $N \times K$ vertices.

Suppose that $V = \{v_1, .., v_{|V|}\}$
Sort $\overline{V} \leftarrow \{v_{i_1}, .., v_{i_{|V|}}\}$ such that $\sigma(v_{i_t}) < \sigma(v_{i_{t+1}})$
Initialize sequence $S \leftarrow \emptyset$
for $v \in \overline{V}$ **do**
    Add $\Omega(v)$ at the end of S
end
return S

2.12 Gated Graph Neural Networks

Long Short-Term Memory (LSTM), firstly proposed by (Hochreiter & Schmidhuber, 1997), is a special kind of Recurrent Neural Network that was designed for learning sequential and time-series data. 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 (Kyunghyun et al., 2014) in their EMNLP 2014 paper in the context of sequential modeling. GRU can be understood as a simplication of LSTM.

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

$$f_v^{(0)} \rightarrow f_v^{(1)} \rightarrow .. \rightarrow f_v^{(L)}$$

in which $f_v^{(\ell)}$ is more global than $f_v^{(\ell-1)}$, and $f_v^{(\ell-1)}$ is more local than $f_v^{(\ell)}$. 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 $f_v^{(\ell)}$ as the final representation or reuse one of the previous level representations $\{f_v^{(0)}, .., f_v^{(\ell-1)}\}$. This idea is inherited from Gated Graph

Neural Networks (GGNN) of (Li et al., 2015) in ICLR 2016. The algorithm (propagation model) of GGNN with GRU is described in pseudocode 13.

Notions:

Algorithm 13: Gated Graph Neural Network

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 $L \in \mathbb{N}$. Each vertex $v \in V$ is associated with an input feature vector x_v.

Result: Construct the corresponding neural network.

Initialize $f_v^{(0)}$ from x_v for all vertex v.
Initialize learnable weight matrices W^z, U^z, W^r, U^r, W and U.
for $\ell = 1 \to L$ **do**
    for $v \in V$ **do**
        $a_v^{(\ell)} = \sum_{v' \in \mathcal{N}(v)} f_{v'}^{(\ell-1)}$
        $z_v^{(\ell)} = \sigma\big(W^z a_v^{(\ell)} + U^z f_v^{(\ell-1)}\big)$
        $r_v^{(\ell)} = \sigma\big(W^r a_v^{(\ell)} + U^r f_v^{(\ell-1)}\big)$
        $\overline{f_v^{(\ell)}} = \tanh\big(W a_v^{(\ell)} + U\big(r_v^{(\ell)} \odot f_v^{(\ell-1)}\big)\big)$
        $f_v^{(\ell)} = (1 - z_v^{(\ell)}) \odot f_v^{(\ell-1)} + z_v^{(\ell)} \odot \overline{f_v^{(\ell)}}$
    end
end
$f_G = \tanh\left(\sum_{v \in V} \sigma\big(i(f_v^{(L)}, x_v)\big) \odot \tanh\big(j(f_v^{(L)}, x_v)\big)\right)$
Use the graph feature f_G for downstream tasks.

- $a_v^{(\ell)}$: Aggregated message of vertex v at level ℓ from its neighborhood vertices $\mathcal{N}(v)$.
- $z_v^{(\ell)}$, $r_v^{(\ell)}$: Forget and update gates of GRU.
- σ: Sigmoid function.
- $\overline{f_v^{(\ell)}}$: Proposed output of vertex v at level ℓ.
- $\odot$: Component-wise multiplication.

- $f_v^{(\ell)}$: Final output (representation) of vertex v at level ℓ.
- f_G: Graph representation.
- $i(.)$, $j(.)$: Multi-layer perceptron.

2.13 Weisfeiler-Lehman Network

Weisfeiler-Lehman Network (WLN) inspired by the Weisfeiler-Lehman isomorphism test for labeled graphs is proposed by (Jin et al., 2017) in their NIPS 2017 paper in the context of molecular graphs. The architecture is designed to embed the computations inherent in WL isomorphism testing to generate learned isomorphism-invariant representations for atoms. Let $c_v^{(L)}$ be the final label of atom a_v where L is the number of levels/layers in WLN. The molecular graph $G = (V, E)$ is represented as a set $\{(c_u^{(L)}, b_{uv}, c_v^{(L)} | (u, v) \in E\}$, where b_{uv} is the bond type between u and v. Let r be the analogous continuous relabeling function. Then a node $v \in G$ with neighbor nodes $\mathcal{N}(v)$, node features f_v, and edge features f_{uv} is relabeled according to:

$$r(v) = \tau\big(U_1 f_v + U_2 \sum_{u \in \mathcal{N}(v)} \tau(V[f_u, f_{uv}])\big) \tag{2.15}$$

where $\tau(.)$ could be any component-wise non-linearity function. We apply this relabeling operation iteratively to obtain context-dependent atom vectors for $1 \leq \ell \leq L$:

$$h_v^{(\ell)} = \tau\big(U_1 h_v^{(\ell-1)} + U_2 \sum_{u \in \mathcal{N}(v)} \tau(V[h_u^{(\ell-1)}, f_{uv}])\big) \tag{2.16}$$

where $h_v^{(0)} = f_v$ and U_1, U_2, V are learnable weight matrices shared across layers. The final atom representations arise from mimicking the set comparison function in the WL

isomorphism test, yielding:

$$c_v = \sum_{u \in \mathcal{N}(v)} W^{(0)} h_u^{(L)} \odot W^{(1)} f_{uv} \odot W^{(2)} h_v^{(L)} \tag{2.17}$$

The set comparison here is realized by matching each rank-1 edge tensor $h_u^{(L)} \otimes f_{uv} \otimes h_v^{(L)}$ to a set of reference edges also cast as rank-1 tensors $W^{(0)}[k] \otimes W^{(1)}[k] \otimes W^{(2)}[k]$, where $W[k]$ is the k-th row of matrix W. In other words, equation 2.17 could be written as:

$$c_v[k] = \sum_{u \in \mathcal{N}(v)} \left\langle W^{(0)}[k] \otimes W^{(1)}[k] \otimes W^{(2)}[k], h_u^{(L)} \otimes f_{uv} \otimes h_v^{(L)} \right\rangle \tag{2.18}$$

The resulting c_v is a vector representation that captures the local chemical environment of the atom (through relabeling) and involves a comparison against a learned set of reference environments. The representation of the whole graph G is simply the sum over all the atom representations, and will be used in downstream regression/classification tasks:

$$c_G = \sum_{v \in V} c_v \tag{2.19}$$

2.14 Message Passing Neural Networks

(Gilmer et al., 2017) reintroduced the existing Message Passing framework in the context of neural networks with an application in quantum chemistry as estimating the solution of Density Functional Theory. Message Passing Neural Networks (MPNNs) operate on undirected graphs G with node features x_v and edge features e_{vw}. The forward pass is divided into two phases: message passing phase and readout phase. In the message passing phase, the iterative algorithm executes for L time steps (or layers) and is defined in terms of message functions M_ℓ and vertex update functions U_ℓ. At layer ℓ, each vertex v is associated with a hidden state (vertex representation) h_v^ℓ and is updated based on the message m_v^ℓ according

to:

$$m_v^{\ell+1} = \sum_{w \in \mathcal{N}(v)} M_\ell(h_v^\ell, h_w^\ell, e_{vw}) \tag{2.20}$$

$$h_v^{\ell+1} = U_\ell(h_v^\ell, m_v^{\ell+1}) \tag{2.21}$$

where $\mathcal{N}(v)$ denotes the neighbors of v in graph G. The readout phase computes a feature vector $\phi(G)$ for the whole graph using some readout function R according to:

$$\phi(G) = R(\{h_v^L | v \in G\}) \tag{2.22}$$

The message functions M_ℓ, vertex update functions U_ℓ, and readout function R are all learned differentiable functions with learnable parameters (e.g., Multi-layer Perceptron or MLP). R operates on the set of vertex representations and is required to be invariant to vertex permutations in order for the MPNN to be permutation-invariant (invariant to graph isomorphism). Many models in the literature (not all) can be projected into the MPNN framework by specifying the message functions M_ℓ, vertex update functions U_ℓ, and readout function R. In addition, MPNN can be easily extended to learn edge representations by introducing hidden states $h_{e_{vw}}^\ell$ for all edges in the graph and updating them analogously to equations 2.20 and 2.21.

2.15 Interaction Networks

Interaction Networks (IN) proposed by (Battaglia et al., 2016) considers the graph learning problem in which each vertex and the whole graph are associated with learning targets. In the language of MPNN framework, the message function $M(h_v, h_w, e_{vw})$ and update function $U(h_v, x_v, m_v)$ of IN are MLPs taking the inputs as a vector concatenation. The graph level output is $R = f(\sum_{v \in G} h_v^L)$ where f is an MLP which takes the sum of the final vertex representation h_v^L. In the original work, the authors only considered the model with $L = 1$.

2.16 Molecular Graph Convolutions

Molecular Graph Convolutions introduced by (Kearns et al., 2016) is based on the MPNN framework in which edge representations are updated during the message passing phase. The vertex update function is:

$$U_\ell(h_v^\ell, m_v^{\ell+1}) = \alpha(W_1(\alpha(W_0 h_v^\ell), m_v^{\ell+1})) \tag{2.23}$$

where $(.,.)$ denotes the concatenation operation, α is the RELU activation. The edge update function is:

$$e_{vw}^{\ell+1} = U'_\ell(e_{vw}^\ell, h_v^\ell, h_w^\ell) = \alpha(W_4(\alpha(W_2 e_{vw}^\ell), \alpha(W_3(h_v^\ell, h_w^\ell)))) \tag{2.24}$$

In both equations 2.23 and 2.24, W_i are learnable weight matrices.

2.17 Deep Tensor Neural Networks

Deep Tensor Neural Networks (DTNN) proposed by (Schütt et al., 2017) focuses on physical systems of particles that can be interpreted as complete graphs. DTNN computes the message from atom w to atom v by:

$$M_\ell = \tanh\left(W^{fc}((W^{cf} h_w^\ell + b_1) \odot (W^{df} e_{vw} + b_2))\right) \tag{2.25}$$

where W^{fc}, W^{cf}, and W^{df} are matrices and b_1, b_2 are bias vectors. The vertex update function is:

$$h_v^{\ell+1} = U_\ell(h_v^\ell, m_v^{\ell+1}) = h_v^\ell + m_v^{\ell+1} \tag{2.26}$$

The readout function is defined as:

$$\phi(G) = \sum_{v \in G} NN(h_v^L) \tag{2.27}$$

where $NN(h_v^L)$ is a single hidden layer neural network (fully-connected) taking input as the hidden state of atom v from the last layer.

2.18 Laplacian Based Methods

Family of Laplacian based models defined in (Defferrard et al., 2016), (Bruna et al., 2014), and (Kipf & Welling, 2017) can be seen as instances of the MPNN framework. These models generalize the notion of convolutions on a general graph G with N vertices. Given an adjacency matrix $A \in \mathbb{R}^{N \times N}$. The graph Laplacian is defined as:

$$L = I - D^{-1/2} A D^{-1/2} \tag{2.28}$$

where D is the diagonal degree matrix. Let V denote the eigenvectors of L, ordered by eigenvalue. Let σ be a real-valued nonlinearity. The fundamental operation in this family of models is the Graph Fourier Transformation. We define an operation which transforms an input vector x of size $N \times d_1$ to an output vector y of size $N \times d_2$ (we transform each vertex representation of size d_1 into size d_2):

$$y_j = \sigma\left(\sum_{i=1}^{d_1} V F_{i,j} V^T x_i \right) \quad (j \in \{1, .., d_2\}) \tag{2.29}$$

The matrices $F_{i,j}$ are all diagonal $N \times N$ matrices and contain all of the learnable parameters in the layer. Define the rank 4 tensor $\hat{L}$ of dimension $N \times N \times d_1 \times d_2$ where $\hat{L}_{v,w,i,j} = (V F_{i,j} V^T)_{v,w}$ in which v, w are the indices corresponding to vertices. Let $\hat{L}_{v,w}$ denote the $d_1 \times d_2$ dimensional matrix where $(\hat{L}_{v,w})_{i,j} = \hat{L}_{v,w,i,j}$. Equation 2.29 can be written as:

$$y_{v,j} = \sigma\left(\sum_{i=1,w=1}^{d_1,N} \hat{L}_{v,w,i,j} x_{w,i} \right) \tag{2.30}$$

or in short:

$$y_{v,:} = \sigma\Big(\sum_{w=1}^{N} \hat{L}_{v,w} x_w \Big) \tag{2.31}$$

We relabel y_v as $h_v^{\ell+1}$ and x_w as h_w^{ℓ}. In the language of the MPNN framework, the message update function can be written as:

$$m_v^{\ell+1} = M(h_v^{\ell}, h_w^{\ell}) = \hat{L}_{v,w} h_w^{\ell} \tag{2.32}$$

and the vertex update function can be written as:

$$h_v^{\ell+1} = U(h_v^{\ell}, m_v^{\ell+1}) = \sigma(m_v^{\ell+1}) \tag{2.33}$$

2.19 Graph-based Semi-Supervised Learning

Graph-based Semi-Supervised Learning (SSL) was proposed by (Ravi & Diao, 2016) in the context of applying streaming approximation algorithm into finding soft assignment of labels in a semi-supervised manner to each vertex in a large-scale graph $G = (V, E, W)$, where V is the set of vertices, E is the set of edges and $W = (w_{uv})$ is the edge weight matrix. Let V_l and V_u be the sets of labeled and unlabeled vertices, respectively. Let $n = |V|$, $n_l = |V_l|$ and $n_u = |V_u|$. We use diagonal matrix S to record the seeds, in which $s_{v,v} = 1$ if the node v is a seed. Let L represent the output label set of size m. Matrix $Y \in \mathbb{R}^{n \times m}$ records the training label distribution for the seeds where $Y_{vl} = 0$ for $v \in V_u$. Matrix $\hat{Y} \in \mathbb{R}^{n \times m}$ is the label distribution assignment matrix for all vertices. The graph-based SSL learns $\hat{Y}$ by propagating the information Y on graph G. To obtain the label distribution $\hat{Y}$, we minimize the following convex objective function:

$$C(\hat{Y}) = \mu_1 \sum_{v \in V_l} s_{vv} \|\hat{Y}_v - Y_v\|_2^2 + \mu_2 \sum_{v \in V, u \in \mathcal{N}(v)} w_{vu} \|\hat{Y}_v - \hat{Y}_u\|^2 + \mu_3 \sum_{v \in V} \|\hat{Y}_v - U\|_2^2 \tag{2.34}$$

with a constraint:

$$\sum_{l=1}^{L} \hat{Y}_{vl} = 1 \quad (\forall v \in V) \tag{2.35}$$

where $\mathcal{N}(v)$ denotes the neighborhood of vertex v, U is the uniform prior distribution over all labels, and μ_1, μ_2, μ_3 are constant hyper-parameters. The objective function 2.34 satisfies the following:

- First term: For all labeled vertices (seeds), the label distribution should be close to the given label assignment.
- Second term: Close vertices should share similar labels.
- Third term: The label distribution should be close to the prior uniform distribution.

In addition, the objective function allows efficient iterative optimization algorithm that is repeated until convergence, in particular Jacobi iterative algorithm which defines the approximate solution at the $(i+1)$-th iteration based on the solution of the i-th iteration:

$$\hat{Y}_{vl}^{(i+1)} = \frac{1}{M_{vl}}(\mu_1 s_{vv} Y_{vl} + \mu_2 \sum_{u \in \mathcal{N}(v)} w_{vu} \hat{Y}_{ul}^{(i)} + \mu_3 U_l) \tag{2.36}$$

$$M_{vl} = \mu_1 s_{vv} + \mu_2 \sum_{u \in \mathcal{N}(v)} w_{vu} + \mu_3 \tag{2.37}$$

where $U_l = 1/m$ which is the uniform distribution on label l. $\hat{Y}_{vl}^{(0)}$ is initialized with seed label weight Y_{vl} if $v \in V_l$, and uniform distribution $1/m$ if $v \in V_u$. This optimization method is called the EXPANDER algorithm.

CHAPTER 3

COVARIANT COMPOSITIONAL NETWORKS FOR LEARNING GRAPHS

This chapter is based on Covariant Compositional Networks presented by (Kondor et al., 2018) and (Hy et al., 2018).

3.1 Compositional Networks

In this section, we introduce a general architecture called **compositional networks** (**compnets**) for representing complex objects as a combination of their parts and show that graph neural networks can be seen as special cases of this framework.

Definition 3.1.1. Let $\mathcal{G}$ be an object with n elementary parts (atoms) $\mathcal{E} = \{e_1, .., e_n\}$. A **compositional scheme** for $\mathcal{G}$ is a directed acyclic graph (DAG) $\mathcal{M}$ in which each node ν is associated with some subset $\mathcal{P}_\nu$ of $\mathcal{E}$ (these subsets are called **parts** of $\mathcal{G}$) in such a way that:

1. In the bottom level, there are exactly n leaf nodes in which each leaf node ν is associated with an elementary atom e. Then $\mathcal{P}_\nu$ contains a single atom e.

2. $\mathcal{M}$ has a unique root node ν_r that corresponds to the entire set $\{e_1, .., e_n\}$.

3. For any two nodes ν and ν', if ν is a descendant of ν', then $\mathcal{P}_\nu$ is a subset of $\mathcal{P}_{\nu'}$.

One can express message passing neural networks in this compositional framework. Consider a graph $G = (V, E)$ in an $L + 1$ layer network. The set of vertices V is also the set of elementary atoms $\mathcal{E}$. Each layer of the graph neural network (except the last) has one node denoted by ν and one feature tensor denoted by f for each vertex of the graph G. The compositional network $\mathcal{N}$ is constructed as follows:

1. In layer $\ell = 0$, each leaf node ν_i^0 represents the single vertex $\mathcal{P}_i^0 = \{i\}$ for $i \in V$. The corresponding feature tensor f_i^0 is initialized by the vertex label l_i.

2. In layers $\ell = 1, 2, .., L$, node ν_i^ℓ is connected to all nodes from the previous level that are neighbors of i in G. The children of ν_i^ℓ are $\{\nu_j^{\ell-1} | j : (i,j) \in E\}$. Thus, $\mathcal{P}_i^\ell = \bigcup_{j:(i,j)\in E} \mathcal{P}_j^{\ell-1}$. The feature tensor f_i^ℓ is computed as an aggregation of feature tensors in the previous layer:

$$f_i^\ell = \Phi(\{f_j^{\ell-1} | j \in \mathcal{P}_i^\ell\})$$

   where Φ is some aggregation function.

3. In layer $\ell = L + 1$, we have a single node ν_r that represents the entire graph and collects information from all nodes at level $\ell = L$:

$$\mathcal{P}_r \equiv V$$

$$f_r = \Phi(\{f_i^L | i \in \mathcal{P}_r\})$$

In the following section, we will refer ν as the **neuron**, and $\mathcal{P}$ and f as its corresponding **receptive field** and **activation**, respectively.

3.2 Covariance

Standard message passing neural networks used summation or averaging operation as the aggregation function Φ of neighboring vertices' feature tensors. That would lead to loss of topological information. Therefore, we propose **permutation covariance** requirement for our neural activations f defined as follows.

Definition 3.2.1. For a graph G with the comp-net $\mathcal{N}$, and an isomorphic graph G' with

comp-net $\mathcal{N}'$, let ν be any neuron of $\mathcal{N}$ and ν' be the corresponding neuron of $\mathcal{N}'$. Assume that $\mathcal{P}_\nu = (e_{p_1}, .., e_{p_m})$ while $\mathcal{P}_{\nu'} = (e_{q_1}, .., e_{q_m})$, and let $\pi \in \mathbb{S}_m$ be the permutation that aligns the orderings of the two receptive fields, i.e., for which $e_{q_{\pi(a)}} = e_{p_a}$. We say that $\mathcal{N}$ is **covariant to permutations** if for any π, there is a corresponding function R_π such that $f_{\nu'} = R_\pi(f_\nu)$.

The definition can be understood as permuting the vertices of graph G will change the activations of its vertices in a manner that is controlled by some fixed function R_π that depends on the permuation π.

3.3 First order Message Passing

We will call the standard message passing as the **zero'th order message passing** in which each vertex is represented by a feature vector of length c (or c channels) which was not expressive enough to capture the structure of its local neighborhood. Hence, we propose **first order message passing** by representing each vertex v by a matrix: $f_v^\ell \in \mathbb{R}^{|\mathcal{P}_v^\ell| \times c}$, each row of this feature matrix corresponds to a vertex in the neighborhood of v.

Definition 3.3.1. We say that ν is a **first order covariant node** in a comp-net if under the permutation of its receptive field $\mathcal{P}_\nu$ by any $\pi \in \mathbb{S}_{|\mathcal{P}_\nu|}$, its activation transforms as $f_\nu \mapsto P_\pi f_\nu$, where P_π is the permutation matrix:

$$[P_\pi]_{i,j} \triangleq \begin{cases} 1, & \pi(j) = i \\ 0, & otherwise \end{cases} \tag{3.1}$$

The transformed activation $f_{\nu'}$ will be:

$$[f_{\nu'}]_{a,s} = [f_\nu]_{\pi^{-1}(a),s}$$

where s is the channel index.

3.4 Second order Message Passing

Instead of representing a vertex with a feature matrix as done in first order message passing, we can represent it by a 3rd order tensor $f_v^\ell \in \mathbb{R}^{|\mathcal{P}_v^\ell|\times|\mathcal{P}_v^\ell|\times c}$ and require these feature tensors to transform covariantly in a similar manner:

Definition 3.4.1. We say that ν is a **second order covariant node** in a comp-net if under the permutation of its receptive field $\mathcal{P}_\nu$ by an $\pi \in \mathbb{S}_{|\mathcal{P}_\nu|}$, its activation transforms as $f_\nu \mapsto P_\pi f_\nu P_\pi^T$. The transformed activation $f_{\nu'}$ will be:

$$[f_{\nu'}]_{a,b,s} = [f_\nu]_{\pi^{-1}(a),\pi^{-1}(b),s}$$

where s is the channel index.

3.5 Third and higher order Message Passing

The similar pattern is applied further for third, forth, and general, k'th order nodes in the comp-net, in which the activations are k'th order tensors, transforming under permutations as $f_\nu \mapsto f_{\nu'}$:

$$[f_{\nu'}]_{i_1,i_2,..,i_k,s} = [f_\nu]_{\pi^{-1}(i_1),\pi^{-1}(i_2),..,\pi^{-1}(i_k),s} \tag{3.2}$$

All but the channel index s (the last index) is permuted when we go from f_ν to $f_{\nu'}$ after some permutation π of the receptive field $\mathcal{P}_\nu$. In general, we will call any quantiy which transforms according to this equation (ignoring the channel index) as a **k'th oder P-tensor**. Since scalars, vectors and matrices can be considered as zeroth, first and second order tensors, respectively, the following definition covers the previous definitions as special cases.

Definition 3.5.1. We say that ν is a **k'th order covariant node** in a comp-net if the

corresponding activation f_ν is a k'th order P-tensor, i.e., it transforms under permutations of $\mathcal{P}_\nu$ according to 3.2.

3.6 Tensor aggregation rules

The previous sections precribed how activations must transform in comp-nets of different orders. Tensor arithmetic provides a compact framework for deriving the general form of the permutation covariant operations. For convenience, we denote tensors as capital letters. Since the activation f is a tensor in general, we will denote it by capital F in the following sections. Recall the four basic operations that can be applied to tensors:

1. The **tensor product** of $A \in \mathcal{T}^k$ with $B \in \mathcal{T}^p$ yields a tensor $C = A \otimes B \in \mathcal{T}^{p+k}$ where:

$$C_{i_1,i_2,..,i_{k+p}} = A_{i_1,i_2,..,i_k} B_{i_{k+1},i_{k+2},..,i_{k+p}}$$

2. The **contraction** of $A \in \mathcal{T}^k$ along the pair of dimensions $\{a, b\}$ (assuming $a < b$) yields a $k-2$ order tensor:

$$C_{i_1,i_2,..,i_k} = \sum_j A_{i_1,..,i_{a-1},j,i_{a+1},..,i_{b-1},j,i_{b+1},..,k}$$

where we assume that i_a and i_b have been removed from the indices of C. Using Einstein notation, this can be written much more compactly as

$$C_{i_1,i_2,..,i_k} = A_{i_1,i_2,..,i_k} \delta^{i_a,i_b}$$

where δ^{i_a,i_b} is the diagonal tensor with $\delta^{i,j} = 1$ if $i = j$ and 0 otherwise. We also generalize contractions to (combinations of) larger sets of indices $\{\{a_1^1, .., a_{p_1}^1\}, \{a_1^2, .., a_{p_2}^2\}, .., \{a_1^q, .., a_{p_q}^q\}\}$

as the $(k - \sum_j p_j)$ order tensor:

$$C_{...} = A_{i_1,i_2,..,i_k} \delta^{a_1^1,..,a_{p_1}^1} \delta^{a_1^2,..,a_{p_2}^2} \cdots \delta^{a_1^q,..,a_{p_q}^q}$$

3. The **projection** of a tensor is defined as a special case of contraction:

$$A \downarrow_{a_1,..,a_p} = A_{i_1,i_2,..,i_k} \delta^{i a_1} \delta^{i a_2} \cdots \delta^{i a_k}$$

where projection of A among indices a_1, .., a_p is denoted as $A \downarrow_{a_1,..,a_p}$.

Proposition 3.6.1 shows that all of the above operations as well as linear combinations preserve permutation covariance property of P-tensors. Therefore, they can be combined within the aggregation function Φ.

Proposition 3.6.1. *Assume that A and B are k'th and p'th order P-tensors, respectively. Then:*

1. *$A \otimes B$ is a $(k+p)$'th order P-tensors.*
2. *$A_{i_1,i_2,..,i_k} \delta^{a_1^1,..,a_{p_1}^1} \cdots \delta^{a_1^q,..,a_{p_q}^q}$ is a $(k - \sum_j p_j)$'th order P-tensor.*
3. *If A_1, .., A_u are k'th order P-tensors and α_1, .., α_u are scalars, then $\sum_j \alpha_j A_j$ is a k'th order P-tensor.*

Propositions 3.6.2, 3.6.3 and 3.6.4 show that tensor promotion, concatenation and production preverse permutation covariance, and hence can be applied within Φ.

Proposition 3.6.2. *Assume that node ν is a descendant of node ν' in a comp-net $\mathcal{N}$. The corresponding receptive fields are $\mathcal{P}_\nu = (e_{p_1}, .., e_{p_m})$ and $\mathcal{P}_{\nu'} = (e_{q_1}, .., e_{q_{m'}})$. Remark that*

$\mathcal{P}_\nu \subseteq \mathcal{P}_{\nu'}$. *Define* $\chi^{\nu\to\nu'} \in \mathbb{R}^{m\times m'}$ *as an indicator matrix:*

$$\chi^{\nu\to\nu'}_{i,j} = \begin{cases} 1, & q_j = p_i \\ 0, & otherwise \end{cases} \tag{3.3}$$

Assume that F_ν *is a* k*'th order P-tensors with respect to permutations* $(e_{p_1}, .., e_{p_m})$*. We have the* ***promoted*** *tensor:*

$$[F_{\nu\to\nu'}]_{i_1,..,i_k} = \chi^{\nu\to\nu'}_{i_1,j_1} \cdots \chi^{\nu\to\nu'}_{i_k,j_k} [F\nu]_{j_1,..,j_k} \tag{3.4}$$

is a k*'th oder P-tensor with respect to permutations of* $(e_{q_1}, .., e_{q_{m'}})$*.*

In equation 3.4, node ν' promotes P-tensors from its children nodes ν with respect to its own receptive field $\mathcal{P}_{\nu'}$ by the appropriate $\chi^{\nu\to\nu'}$ matrix such that all promoted tensors $F_{\nu\to\nu'}$ have the same size. Remark that promoted tensors are padded with zeros.

Proposition 3.6.3. *Let nodes* ν_1*, ..,* ν_n *be the children of* ν *in a message passing type comp-net (the corresponding vertices of these nodes are in* $\mathcal{P}_\nu$*) with corresponding* k*'th order tensor activations* F_{ν_1}*, ..,* F_{ν_n}*. Let*

$$[F_{\nu_t\to\nu}]_{i_1,..,i_k} = [\chi^{\nu_t\to\nu}]_{i_1,j_1} \cdots [\chi^{\nu_t\to\nu}]_{i_k,j_k} [F_{\nu_t}]_{j_1,..,j_k}$$

be the promoted tensors ($t \in \{1, .., n\}$*). We* ***concatenate*** *or* ***stack*** *them into a* $(k+1)$*'th order tensor:*

$$[\overline{F}\nu]_{t,i_1,..,i_k} = [F_{\nu_t\to\nu}]_{i_1,..,i_k}$$

Then the concatenated tensor $\overline{F}_\nu$ *is a* $(k+1)$*'th order P-tensor of* ν*.*

The restriction of the adjacency matrix to $\mathcal{P}_\nu$ is a second order P-tensor. Proposition 3.6.4 gives us a way to explicitly add topolocical information to the activation.

Proposition 3.6.4. *If F_ν is a k'th order P-tensor at node ν, and $A\downarrow_{\mathcal{P}_\nu}$ is the restriction of the adjacency matrix to $\mathcal{P}_\nu$, then $F_\nu \otimes A\downarrow_{\mathcal{P}_\nu}$ is a $(k+2)$'th order P-tensor.*

3.7 Second order tensor aggregation with the adjacency matrix

The first nontrivial tensor contraction case occurs when $F_{\nu_1\to\nu}$, .., $F_{\nu_n\to\nu}$ are second order tensors, and we multiply with $A\downarrow_{\mathcal{P}_\nu}$, since in that case $\mathcal{T}$ is 5th order (6th order if we consider the channel index), and can be contracted down to second order in the following ways:

1. The **1+1+1** case contracts $\mathcal{T}$ in the form $\mathcal{T}_{i_1,i_2,i_3,i_4,i_5}\delta^{i_{a_1}}\delta^{i_{a_2}}\delta^{i_{a_3}}$, i.e., it projects $\mathcal{T}$ down along 3 of its 5 dimensions. This can be done in $\binom{5}{3} = 10$ ways.

2. The **1+2** case contracts $\mathcal{T}$ in the form $\mathcal{T}_{i_1,i_2,i_3,i_4,i_5}\delta^{i_{a_1}}\delta^{i_{a_2},i_{a_3}}$, i.e., it projects $\mathcal{T}$ along one dimension, and contracts it along two others. This can be done in $3\binom{5}{3} = 30$ ways.

3. The **3** case is a single 3-fold contraction $\mathcal{T}_{i_1,i_2,i_3,i_4,i_5}\delta^{i_{a_1},i_{a_2},i_{a_3}}$. This can be done in $\binom{5}{3} = 10$ ways.

Totally, we have 50 different contractions that result in 50 times more channels. In practice, we only implement 18 contractions for efficiency.

3.8 Architecture

In this section, we will describe how our compositional architecture is a generalization of previous works with an extension to higher-order representations.

Recent works on graph neural networks Duvenaud et al. (2015); Kipf & Welling (2017); Li et al. (2015); Gilmer et al. (2017) can all be seen as instances of *zeroth order message passing* where each vertex representation is a vector (1st order tensor) of c channels in which

each channel is represented by a scalar (zeroth order P-tensor). This results in the loss of certain structural information during the message aggregation, and the network loses the ability to learn topological information of the graph's multiscale structure.

Our architecture represents generalized vertex representations with higher-order tensors which can retain this structural information. There is significant freedom in the choice of this tensor structure, and we now explore two examples, corresponding to the tensor structures, which we call "first order CCN" and "second order CCN", respectively.

We start with an input graph $G = (V, E)$ and construct a network with $L+1$ levels, indexed from 0 (input level) to L (top level). Initially, each vertex v is associated with an input feature vector $l_v \in \mathbb{R}^c$ where c denotes the number of channels. The receptive field of vertex v at level ℓ is denoted by $\mathcal{P}_v^\ell$ and is defined recursively as follows:

$$\mathcal{P}_v^\ell \triangleq \begin{cases} \{v\}, & \ell = 0 \\ \bigcup\limits_{(u,v)\in E} \mathcal{P}_u^\ell, & \ell = 1, \ldots, L \end{cases} \tag{3.5}$$

The vertex representation of vertex v at level ℓ is denoted by a feature tensor F_v^ℓ. In zeroth order message passing, $F_v^\ell \in \mathbb{R}^c$ is a vector of c channels. Let N be the number of vertices in $\mathcal{P}_v^\ell$. In the first order CCN, each vertex is represented by a matrix (second order tensor) $F_v^\ell \in \mathbb{R}^{N\times c}$ in which each row corresponds to a vertex in the receptive field $\mathcal{P}_v^\ell$, and each channel is represented by a vector (first order P-tensor) of size N. In the second order CCN, F_ℓ^v is promoted into a third order tensor of size $N \times N \times c$ in which each channel has a second order representation (second order P-tensor). In general, we can imagine a series of feature tensors of increasing order for higher order message passing. Note that the components corresponding to the channel index does not transform as a tensor, whereas the

remaining indices do transform as a P-tensor. The tensor F_v^ℓ transforms in a *covariant* way with respect to the permutation of the vertices in the receptive field $\mathcal{P}_v^\ell$.

Now that we have established the structure of the high order representations of the vertices at each site, we turn to the task of constructing the aggregation function Φ from one level to another. The key to doing this in a way that preserves covariance is to "promote-stack-reduce" the tensors as one traverses the network at each level.

We start with the promotion step. Recall that we want to accumulate information at higher levels based upon the receptive field of a given vertex. However, it is clear that not all vertices in the receptive field have the same size tensors. To account for this, we use an index function χ that ensures all tensors are the same size by padding with zeros when necessary. At level ℓ, given two vertices v and w such that $\mathcal{P}_w^{\ell-1} \subseteq \mathcal{P}_v^\ell$, the permutation matrix $\chi_\ell^{w\to v}$ of size $|\mathcal{P}_v^\ell| \times |\mathcal{P}_w^{\ell-1}|$ is defined as in Prop. 3.6.2. In CCN 1D & 2D, the resizing is done by (broadcast) matrix multiplication $\chi \cdot F_w^{\ell-1}$ and $\chi \times F_w^{\ell-1} \times \chi^T$ where $\chi = \chi_\ell^{w\to v}$, respectively. Denote the resized tensor as $F_{w\to v}^\ell$. (See step 7 in algorithm 15.) This promotion is done for all tensors of every vertex in the receptive field, and stacked/concatenated into a tensor one order higher. (See Prop. 3.6.3. Notice that the stacked index has the same size as the receptive field.) From here, as in CCN 2D, we can compute the *tensor product* of this higher order tensor with the restricted adjacency matrix (subject to the receptive field) and obtain an even higher order tensor. (See Prop. 3.6.4.) Finally, we can reduce the higher order tensor down to the expected size of the vertex representation using the tensor contractions discussed in Prop. 3.6.1.

We include all possible tensor contractions, which introduces additional channels. To avoid an exponential explosion in the number of channels with deep networks, we use a learnable

set of weights that reduces the number of channels to a fixed number c. These weights are learnable through backpropagation. To complete our construction of Φ, this tensor is passed through an element-wise nonlinear function Υ such as a ReLU to form the feature tensor for a given vertex at the next level. (See steps 4 and 9 in algorithm 15.)

Finally, at the output of the network, we again reduce the vertex representations F_v^ℓ into a vector of channels $\Theta(F_v^\ell) = F_v^\ell \downarrow_{i_1,..,i_p}$ where $i_1, .., i_p$ are the non-channel indices. (See Prop. 3.6.1.) We sum up all the reduced vertex representations of a graph to get a single vector which we use as the graph representation. This final graph representation can then be used for regression or classification with a fully connected layer. In addition, we can construct a richer graph representation by concatenating the shrunk representation at each level. (See steps 12, 13 and 14 in algorithm 15.)

The development of higher order CCNs require efficient tensor algorithms to successively train the network. A fundamental roadblock we face in implementing CCNs is that fifth or sixth order tensors are often too large to be held in memory. To address this challenge, we do not construct the tensor product explicitly. Instead we introduce a *virtual indexing system* for a *virtual tensor* that computes the elements of tensor only when needed given the indices. This allows us to implement the tensor contraction operations efficiently with GPUs on virtual tensors.

For example, consider the operations in step 8 in algorithm 15. This requires performing contractions over several indices on the two inputs $\mathcal{F} = \{F_{w\to v}^\ell | w \in \mathcal{P}_v^\ell\}$, in which $\mathcal{F}_{i_1} = F_{w_{i_1}\to v}^\ell$ is of size $|\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell| \times c$, and $\mathcal{A} = A \downarrow_{\mathcal{P}_v^\ell}$. One naive strategy would be to stack all tensors in $\mathcal{F}$ into a new object and then directly compute the tensor product with $\mathcal{A}$ to form a sixth order tensor, given by a tuple of indices $(i_1, i_2, i_3, i_4, i_5, i_6)$. Instead, the

corresponding tensor element is computed on-the-fly through simple multiplication:

$$\mathcal{T}_{i_1,i_2,i_3,i_4,i_5,i_6} = (\mathcal{F}_{i_1})_{i_2,i_3,i_6} \cdot \mathcal{A}_{i_4,i_5} \tag{3.6}$$

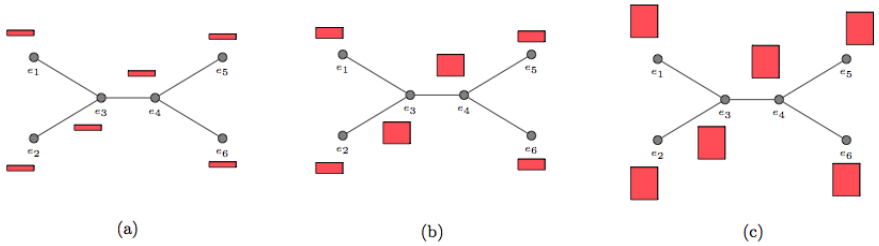
where i_6 is the channel index.

In our experiments of CCN 2D, we implement 18 different contractions (see Prop. 3.7) such that each contraction results in a $|\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell| \times c$ tensor. The result of step 8 is $\overline{F}_\ell^v$ with 18 times more channels.

Algorithm CCN 1D is described in pseudocode 14. Figure 3.1 shows a visualization of CCN 1D's tensors on C_2H_4 molecular graph. Vertices e_3 and e_4 are carbon (C) atoms, and vertices e_1, e_2, e_5 and e_6 are hydrogen (H) atoms. Edge (e_3, e_4) is a double bond (C, C) between two carbon atoms. All other edges are single bonds (C, H) between a carbon atom and a hydrogen atom. In the bottom layer $\ell = 0$, the receptive field of every atom e only contains itself, thus its representation F_e^0 is a tensor of size $1 \times c$ where c is the number of channels (see figure 3.1(a)). In the first layer $\ell = 1$, the receptive field of a hydrogen atom contains itself and the neighboring carbon atom (i.e., $\mathcal{P}_{e_1}^1 = \{e_1, e_3\}$), thus tensors for hydrogen atoms are of size $2 \times c$. Meanwhile, the receptive field of a carbon atom contains itself, the another carbon and two other neighboring hydrogens (i.e., $\mathcal{P}_{e_3}^1 = \{e_1, e_2, e_3, e_4\}$) and $\mathcal{P}_{e_4}^1 = \{e_3, e_4, e_5, e_6\}$), thus $F_{e_3}^1, F_{e_4}^1 \in \mathbb{R}^{4\times c}$ (see figure 3.1(b)). In all later layers denoted $\ell = \infty$, the receptive field of every atom contains the whole graph (in this case, 6 vertices in total), thus $F_e^\infty \in \mathbb{R}^{6\times c}$ (see figure 3.1(c)).

Algorithm CCN 2D is described in pseudocode 15. Figure 3.2 shows a visualization of CCN 2D's tensors on C_2H_4 molecular graph. In the bottom layer $\ell = 0$, $|\mathcal{P}_e^0| = 1$ and

Figure 3.1: CCN 1D on C_2H_4 molecular graph



Algorithm 14: First-order CCN

Input: G, l_v, L

Parameters: Matrices $W_0 \in \mathbb{R}^{c\times c}$, $W_1, .., W_L \in \mathbb{R}^{(2c)\times c}$ and biases $b_0, .., b_L$. For CCN 1D, we only implement 2 tensor contractions.

$F_v^0 \leftarrow \Upsilon(W_0 l_v + b_0 \mathbb{1})$ $(\forall v \in V)$

Reshape F_v^0 to $1 \times c$ $(\forall v \in V)$

for $\ell = 1, .., L$ **do**

  for $v \in V$ **do**

    $F_{w\to v}^{\ell} \leftarrow \chi \times F_w^{\ell-1}$ where $\chi = \chi_{w\to v}^{\ell}$ $(\forall w \in \mathcal{P}_v^{\ell})$

    Concatenate the promoted tensors in $\{F_{w\to v}^{\ell} | w \in \mathcal{P}_v^{\ell}\}$ and apply 2 tensor contractions that results in $\overline{F}_v^{\ell} \in \mathbb{R}^{|\mathcal{P}_v^{\ell}|\times(2c)}$.

    $F_v^{\ell} \leftarrow \Upsilon(\overline{F}_v^{\ell} \times W_{\ell} + b_{\ell}\mathbb{1})$

  end

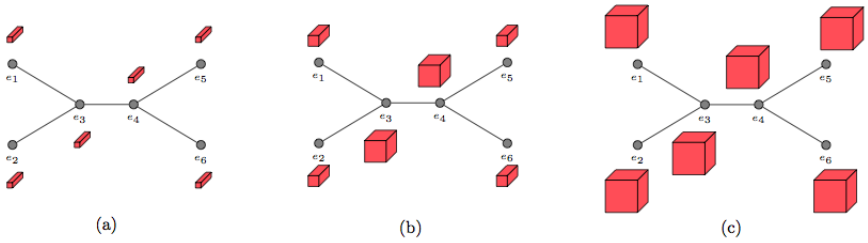
end

$F^{\ell} \leftarrow \sum_{v\in V} \Theta(F_v^{\ell})$ $(\forall \ell)$

Graph feature $F \leftarrow \bigoplus_{\ell=0}^{L} F^{\ell} \in \mathbb{R}^{(L+1)c}$

Use F for downstream tasks.

Figure 3.2: CCN 2D on C_2H_4 molecular graph



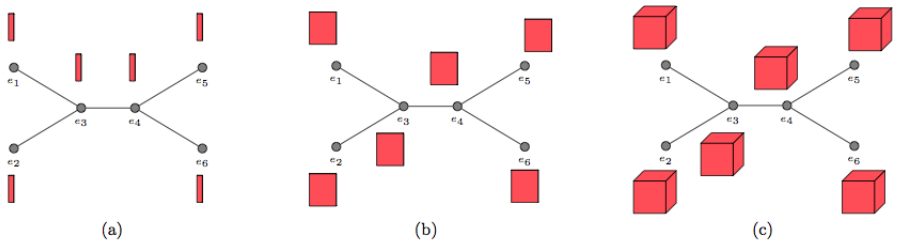
$F_e^0 \in \mathbb{R}^{1\times1\times c}$ for every atom e (see figure 3.2(a)). In the first layer $\ell = 1$, $|\mathcal{P}_e^1| = 2$ and $F_e^1 \in \mathbb{R}^{2\times2\times c}$ for hydrogen atom $e \in \{e_1, e_2, e_5, e_6\}$, and for carbon atoms $|\mathcal{P}_{e_3}^1| = |\mathcal{P}_{e_4}^1| = 4$ and $F_{e_3}^1, F_{e_4}^1 \in \mathbb{R}^{4\times4\times c}$ (see figure 3.2(b)). In all other layers $\ell = \infty$, $\mathcal{P}_e^\infty \equiv V$ and $F_e^\infty \in \mathbb{R}^{6\times6\times c}$ $(\forall e)$ (see figure 3.2(c)).

Algorithm 15: Second-order CCN

Input: G, l_v, L
Parameters: Matrices $W_0 \in \mathbb{R}^{c\times c}$, $W_1, .., W_L \in \mathbb{R}^{(18c)\times c}$ and biases $b_0, .., b_L$.
$F_v^0 \leftarrow \Upsilon(W_0 l_v + b_0 \mathbb{1})$ $(\forall v \in V)$
Reshape F_v^0 to $1 \times 1 \times c$ $(\forall v \in V)$
for $\ell = 1, .., L$ **do**
  for $v \in V$ **do**
    $F_{w\to v}^\ell \leftarrow \chi \times F_w^{\ell-1} \times \chi^T$ where $\chi = \chi_{w\to v}^\ell$ $(\forall w \in \mathcal{P}_v^\ell)$
    Apply virtual tensor contraction algorithm (Sec.3.7) with inputs $\{F_{w\to v}^\ell | w \in \mathcal{P}_v^\ell\}$ and the restricted adjacency matrix $A\downarrow_{\mathcal{P}_v^\ell}$ to compute $\overline{F}_v^\ell \in \mathbb{R}^{|\mathcal{P}_v^\ell|\times|\mathcal{P}_v^\ell|\times(18c)}$.
    $F_v^\ell \leftarrow \Upsilon(\overline{F}_v^\ell \times W_\ell + b_\ell \mathbb{1})$
  end
end
$F^\ell \leftarrow \sum_{v\in V} \Theta(F_v^\ell)$ $(\forall \ell)$
Graph feature $F \leftarrow \bigoplus_{\ell=0}^{L} F^\ell \in \mathbb{R}^{(L+1)c}$
Use F for downstream tasks.

Figure 3.3 shows the difference among zeroth, first and second order message passing (see from left to right) with layer $\ell \geq 2$. In the zeroth order, the vertex representation is always a vector of c channels (see figure 3.3(a)). In the first and second order (see figures

Figure 3.3: Zeroth, first and second order message passing



3.3(b)(c)), the vertex representation is a matrix of size $6 \times c$ or a 3rd order tensor of size $6 \times 6 \times c$ in which each channels is represented by a vector of length 6 or a matrix of size 6×6, respectively. With higher orders, CCNs can capture more topological information.

CHAPTER 4

GRAPHFLOW DEEP LEARNING FRAMEWORK IN C++/CUDA

4.1 Motivation

Many Deep Learing frameworks have been proposed over the last decade. Among them, the most successful ones are TensorFlow Abadi et al. (2016), PyTorch Paszke et al. (2017), Mxnet Chen et al. (2016), Theano Al-Rfou et al. (2016). 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:

1. The current frameworks are not flexible for an implementation of the **Virtual Indexing System** (see section 3.8) for efficient and low-cost tensor operations.

2. 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 Looks et al. (2017).

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.

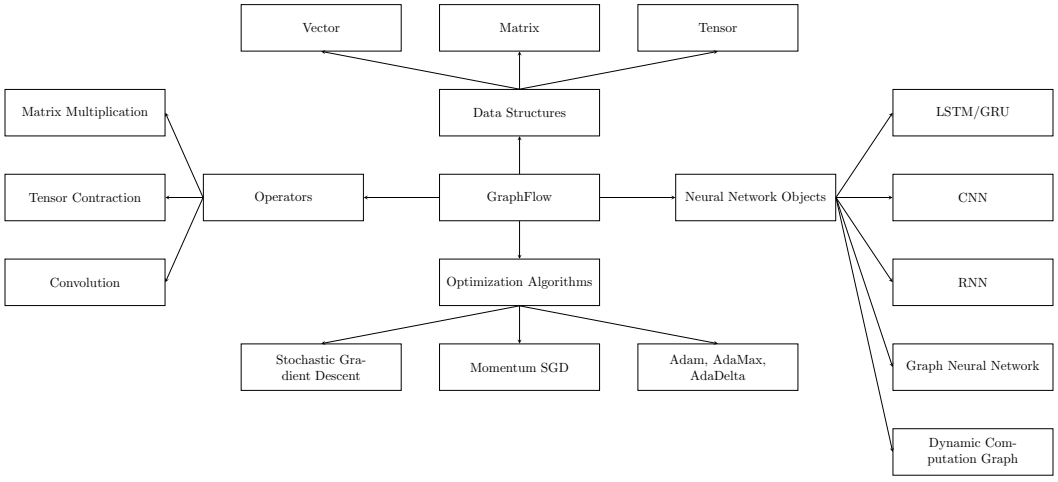
Figure 4.1 describes the general structure of GraphFlow Deep Learning framework.

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

Figure 4.1: GraphFlow overview



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:

$$F_v^\ell \otimes A \downarrow_{\mathcal{P}_v^\ell}$$

where $F_v^\ell \in \mathbb{R}^{|\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell| \times c}$ is the result from tensor stacking operation of vertex v at level ℓ, $A \downarrow_{\mathcal{P}_v^\ell} \in \{0,1\}^{|\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell|}$ is the restricted adjacency matrix to the receptive field $\mathcal{P}_v^\ell$, and c is the number of channels. 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 $F_v^\ell \otimes A \downarrow_{\mathcal{P}_v^\ell}$ of 6th order into 3rd order tensor of size $|\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell| \times c$. As discussed section 3.7, there are 18 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, F_v^ℓ is stored as a float array of size $|\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell| \times c$, and the reduced adjacency matrix $A \downarrow_{\mathcal{P}_v^\ell}$ is stored as a float array of size $|\mathcal{P}_v^\ell| \times |\mathcal{P}_v^\ell|$. We divide the job to GPU in such a way that each thread processes a part of F_v^ℓ and a part of $A \downarrow_{\mathcal{P}_v^\ell}$. 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.

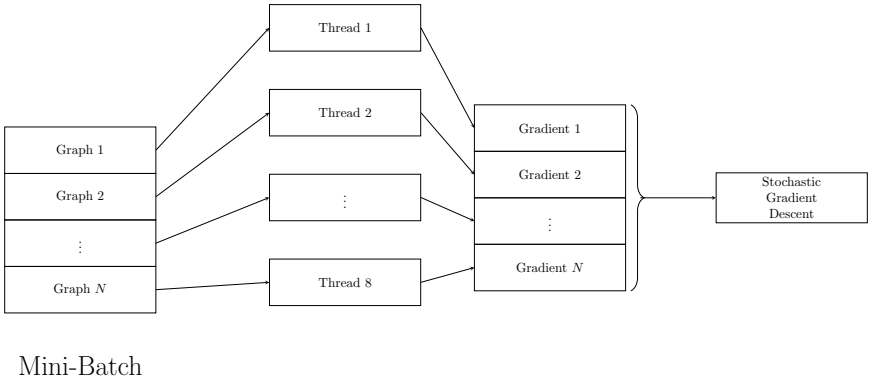
Technically, suppose that we can execute T threads concurrent at a time for gradient computation jobs. Before every training starts, we initialize exactly T identical dynamic computation graphs by GraphFlow. Given a minibatch of M training examples, we distribute the examples to T thread, each thread 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 4.2.

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 computation. This could prevent us from achieving high efficient computation since data movement takes time and the GPU cannot be used until this

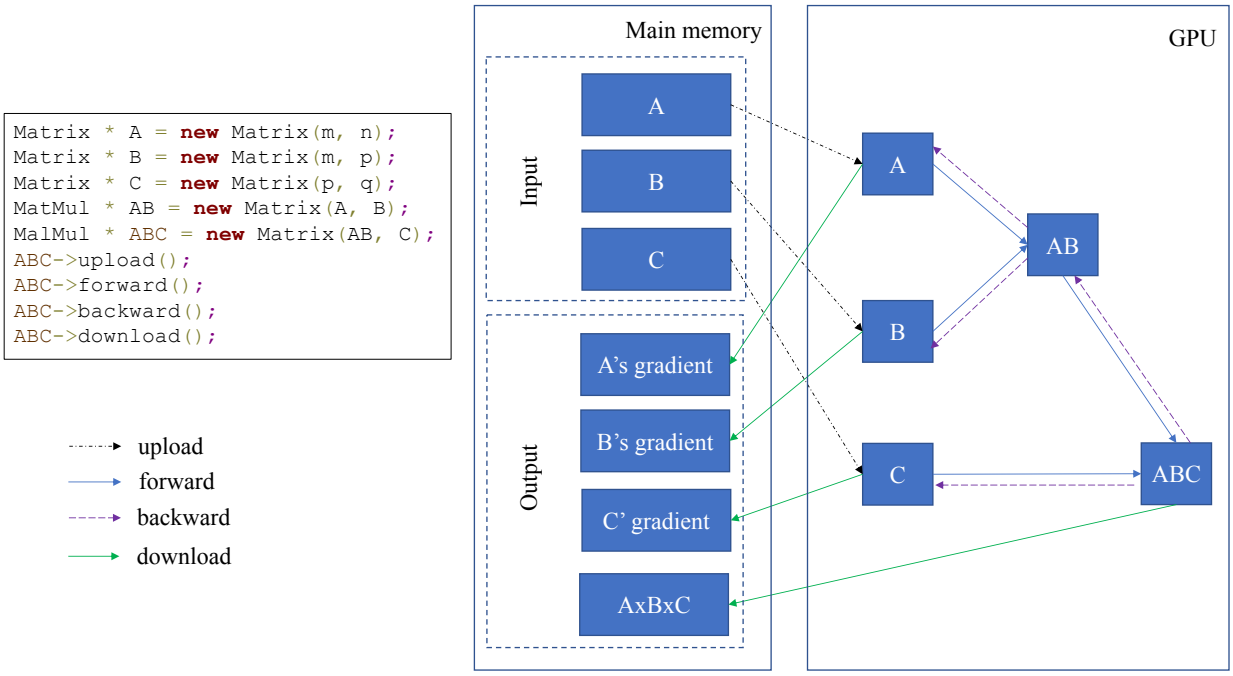
Figure 4.2: CPU multi-threading for gradient computation



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 4.3 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.

Figure 4.3: Example of data flow between GPU and main memory



4.3.6 Source code

The source code of GraphFlow Deep Learning framework can be found at:

`https://github.com/HyTruongSon/GraphFlow`

CHAPTER 5

EXPERIMENTS AND RESULTS

5.1 Efficiency of GraphFlow framework

5.1.1 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. Table 5.1 and figure 5.1 show the details.

5.1.2 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 $O(18 \times |\mathcal{P}_v^\ell|^5 \times c)$ that grows exponentially with the size of receptive field $|\mathcal{P}_v^\ell|$ and grows linearly with the number of channels c. We have a constant 18 as the number of contractions implemented in the second-order case. We have several tests with the size of receptive field $|\mathcal{P}_v^\ell|$ ranging in $\{5, 10, 20, 35\}$ and the number of channels c ranging in $\{10, 20\}$. In the largest case with $|\mathcal{P}_v^\ell| = 35$ and $c = 20$, we observe that GPU gives a factor of approximately 62x speedup. Table 5.2 and figure 5.2 show the details. Figure 5.3 describes the general idea of GPU implementation of tensor contractions in CCN 2D by **Virtual Indexing System**.

Table 5.1: GPU vs CPU matrix multiplication running time (milliseconds)

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

Figure 5.1: GPU vs CPU matrix multiplication running time (milliseconds) in $\log_{10}$ scale

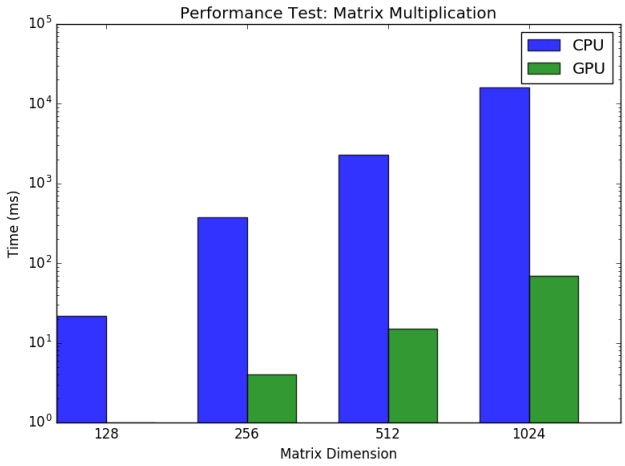


Table 5.2: GPU vs CPU tensor contraction running time (milliseconds)

$\lvert\mathcal{P}_v^\ell\rvert$	c	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

Figure 5.2: GPU vs CPU tensor contraction running time (milliseconds) in $\log_{10}$ scale

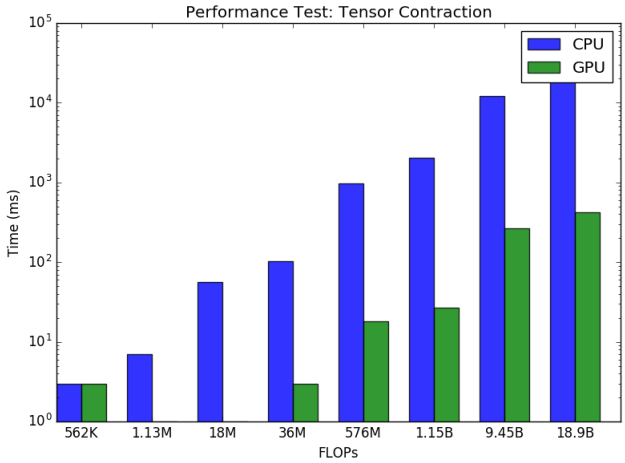


Figure 5.3: GPU implementations of tensor contractions in CCN 2D

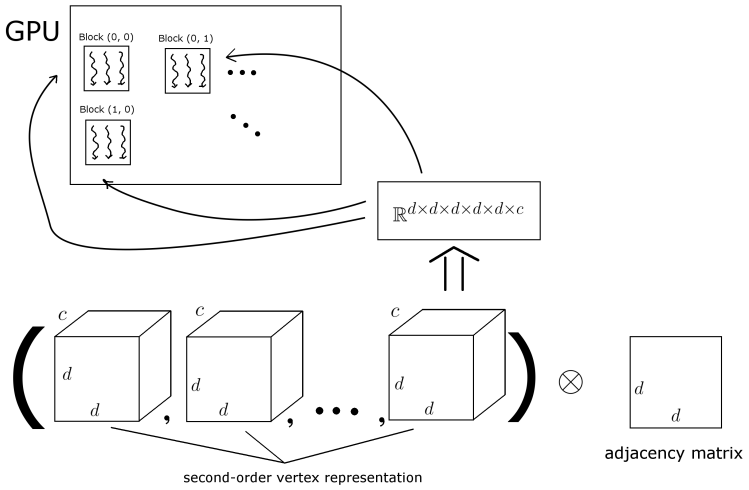


Table 5.3: GPU and CPU network evaluation running time (milliseconds)

$\|V\|$	Max $\|\mathcal{P}_v^\ell\|$	c	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

5.1.3 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 $|\mathcal{P}_v^\ell|$ as 10 and 15, the number of channels c 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. Table 5.3 shows the detail.

5.1.4 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,

Table 5.4: Single thread vs Multiple threads running time

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

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 $c = 10$ 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 5.4 shows the details.

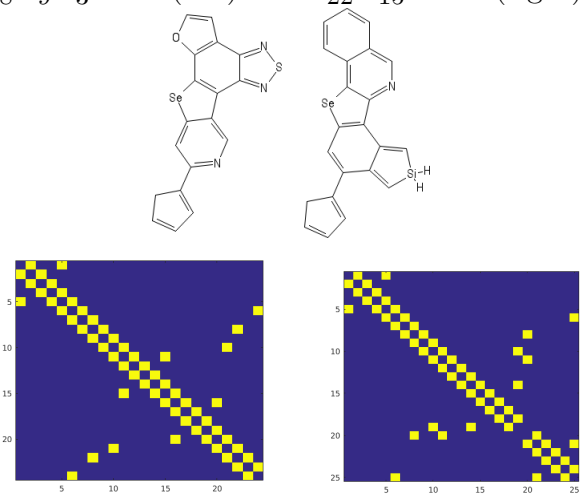
5.2 Experiments

We now compare our CCN framework (Section 3.8) to several standard graph learning algorithms. We focus on two datasets that contain the result of a large number of Density Functional Theory (DFT) calculations:

1. **The Harvard Clean Energy Project (HCEP)**, consisting of 2.3 million organic compounds that are candidates for use in solar cells (Hachmann et al., 2011). Figure 5.4 shows an example of two molecules with similar graph structures in the HCEP dataset.

2. **QM9**, a dataset of ∼134k organic molecules with up to nine heavy atoms (C, O, N and F) (Ramakrishnan et al., 2014) out of the GDB-17 universe of molecules (Ruddigkeit et al., 2012). Each molecule contains data including 13 target chemical properties, along with the spatial position of every constituent atom.

Figure 5.4: Molecules $C_{18}H_9N_3OSSe$ (left) and $C_{22}H_{15}NSeSi$ (right) with adjacency matrices



DFT (Hohenberg & Kohn, 1964; Kohn & Sham, 1965) is the workhorse of the molecular chemistry community, given its favorable tradeoff between accuracy and computational power. Still, it is too costly for tasks such as drug discovery or materials engineering, which may require searching through millions of candidate molecules. An accurate prediction of molecular properties would significantly aid in such tasks.

We are interested in the ability of our algorithm to learn on both pure graphs, and also on physical data. As such, we perform three experiments. We start with two experiments based only upon atomic identity and molecular graph topology:

1. **HCEP**: We use a random sample of 50,000 molecules of the HCEP dataset; our learning target is Power Conversion Efficiency (PCE), and we present the mean average error (MAE). The input vertex feature l_v is a one-hot vector of atomic identity concatenated with purely synthesized graph-based features.

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

We also tested our algorithm's ability to learn on DFT data based upon physical features. We perform the following experiment:

3. **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. (See Fig. 5.9). 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. (See step 12 of the algorithm 15).

   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 (Faber et al., 2017).

In the case of HCEP, we compared CCNs to lasso, ridge regression, random forests, gradient boosted trees, optimal assignment Weisfeiler–Lehman graph kernel (Kriege et al., 2016) (WL), neural graph fingerprints (Duvenaud et al., 2015), and the "patchy-SAN" convolutional type algorithm (referred to as PSCN) (Niepert et al., 2016). For the first four of

these baseline methods, we created simple feature vectors from each molecule: the number of bonds of each type (i.e., number of H–H bonds, number of C–O bonds, etc.) and the number of atoms of each type. Molecular graph fingerprints uses atom labels of each vertex as base features. For ridge regression and lasso, we cross validated over λ. For random forests and gradient boosted trees, we used 400 trees, and cross validated over max depth, minimum samples for a leaf, minimum samples to split a node, and learning rate (for GBT). For neural graph fingerprints, we used 3 layers and a hidden layer size of 10. In PSCN, we used a patch size of 10 with two convolutional layers and a dense layer on top as described in their paper.

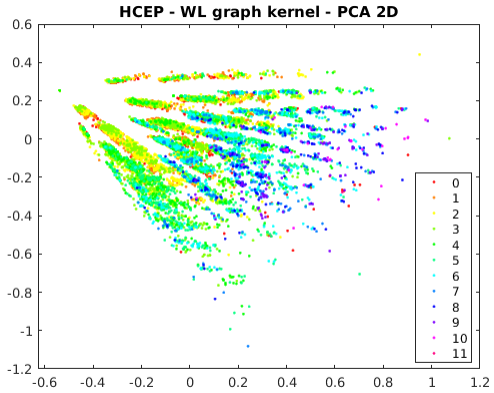
For QM9(a), we compared against the Weisfeiler–Lehman graph kernel, neural graph fingerprints, and PSCN. The settings for NGF and PSCN are as described for HCEP. For QM9(b), we compared against DFT error provided in (Faber et al., 2017).

We initialized the synthesized graph-based features of each vertex with computed histogram alignment features, inspired by (Kriege et al., 2016), of depth up to 10. Each vertex receives a base label $l_v = \text{concat}_{d=1}^{10} H_v^d$ where $H_v^d \in \mathbb{R}^c$ (with c being the total number of distinct discrete node labels) is the vector of relative frequencies of each label for the set of vertices at distance equal to d from vertex v. Our CCNs architecture contains up to five levels.

In each experiment we separated 80% of the dataset for training, 10% for validation, and evaluated on the remaining 10% test set. We used Adam optimization (Kingma & Ba, 2015) with the initial learning rate set to 0.001 after experimenting on a held out validation set. The learning rate decayed linearly after each step towards a minimum of 10^{-6}.

Our method, Covariant Compositional Networks, and other graph neural networks such

Figure 5.5: 2D PCA projections of Weisfeiler-Lehman features in HCEP



as Neural Graph Fingerprints (Duvenaud et al., 2015), PSCN (Niepert et al., 2016) and Gated Graph Neural Networks (Li et al., 2015) are implemented based on the GraphFlow framework (see chapter 4).

Tables 5.5, 5.6, and 5.7 show the results of HCEP, QM9(a) and QM9(b) experiments, respectively. Figures 5.5 and 5.6 show the 2D PCA projections of *learned* molecular representations in HCEP dataset with Weisfeiler-Lehman, Covariant Compositional Networks 1D & 2D, respectively. On the another hand, figures 5.7 and 5.8 show the 2D projections with t-SNE (Maaten & Hinton, 2008). The colors represent the PCE values ranging from 0 to 11. Figure 5.10 shows the distributions between ground-truth and prediction of CCN 1D & 2D in HCEP.

5.2.1 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).

Figure 5.6: 2D PCA projections of CCNs graph representations in HCEP

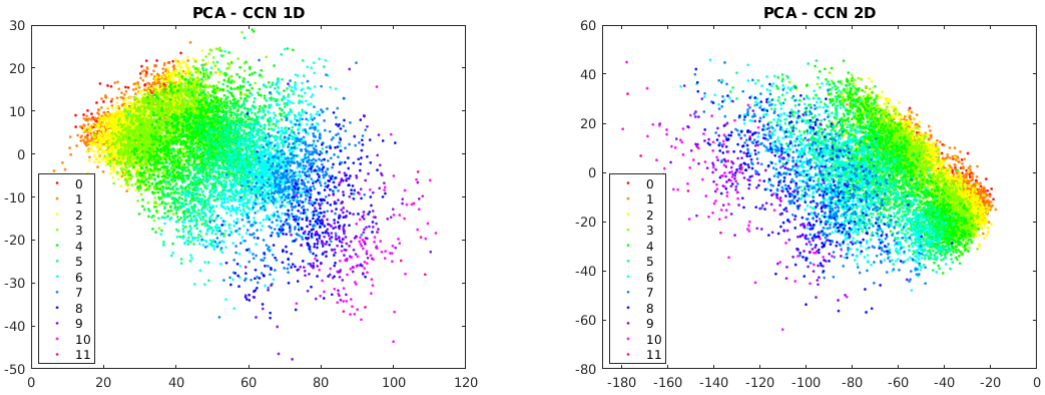


Figure 5.7: 2D t-SNE projections of Weisfeiler-Lehman features in HCEP

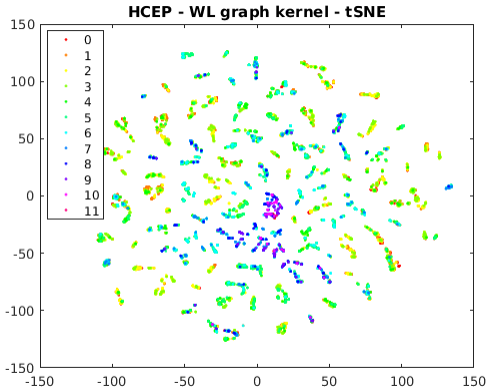
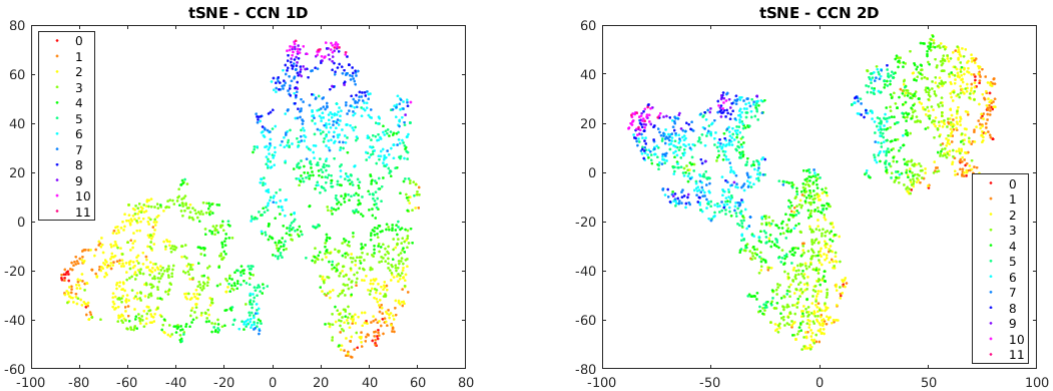


Figure 5.8: 2D t-SNE projections of CCNs graph representations in HCEP



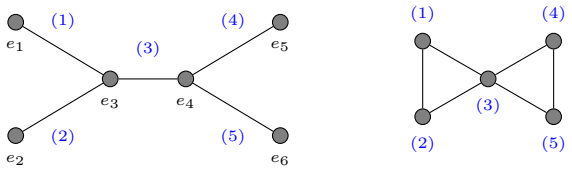


Figure 5.9: Molecular graph of C_2H_4 (left) and its corresponding line graph (right).

Table 5.5: HCEP regression results

	Test MAE	Test RMSE
Lasso	0.867	1.437
Ridge regression	0.854	1.376
Random forest	1.004	1.799
Gradient boosted trees	0.704	1.005
WL graph kernel	0.805	1.096
Neural graph fingerprints	0.851	1.177
PSCN	0.718	0.973
CCN 1D	**0.216**	**0.291**
CCN 2D	**0.340**	**0.449**

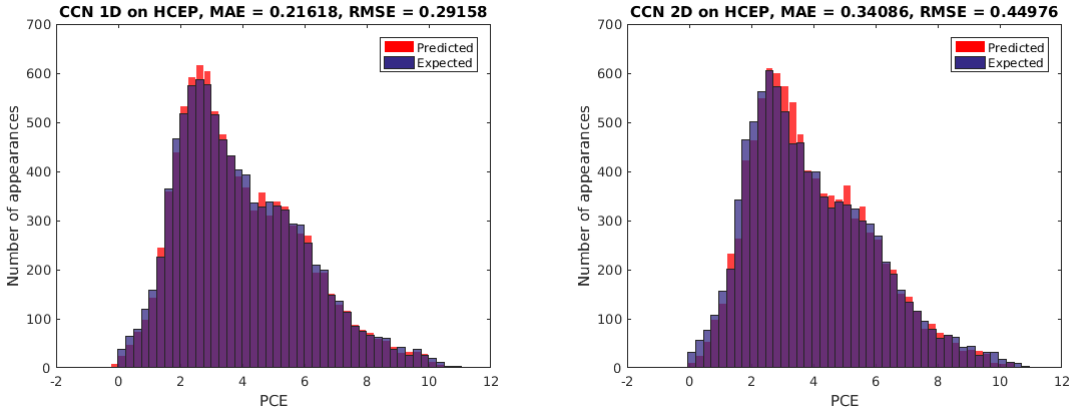
Table 5.6: QM9(a) regression results (MAE)

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 5.7: 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

Figure 5.10: Distributions of ground-truth and prediction of CCN 1D & 2D in HCEP



CHAPTER 6

CONCLUSION AND FUTURE RESEARCH

We extended Message Passing Neural Networks 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 other 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. The next step would be to find applications of CCNs in different areas of computer science, for example applying graph neural networks on large-scale data center's network topology and monitoring timeseries data to detect and find the root causes of network failures. We are developing our custom Deep Learning framework in C++/CUDA named **GraphFlow** which supports automatic and symbolic differentitation, dynamic computation graph as well as complex tensor/matrix operations 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.

REFERENCES

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, and X. Zheng Y. Yu. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *https://arxiv.org/abs/1603.04467*, 2016.

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, and X. Glorot. Theano: A python framework for fast computation of mathematical expressions. *https://arxiv.org/abs/1605.02688*, 2016.

L. Babai and L. Kucera. Canonical labelling of graphs in linear average time. *Proceedings Symposium on Foundations of Computer Science*, pp. 39–46, 1979.

P. W. Battaglia, R. Pascanu, M. Lai, D. Rezende, and K. Kavukcuoglu. Interaction networks for learning about objects, relations and physics. *Advances in Neural Information Processing Systems (NIPS)*, pp. 4502–4510, 2016.

K. M. Borgwardt and H. P. Kriegel. Shortest-path kernels on graphs. In *Proceedings of the 5th IEEE International Conference on Data Mining(ICDM) 2005), 27-30 November 2005, Houston, Texas, USA*, pp. 74–81, 2005.

R. D. L. Briandais. File searching using variable length keys. *Proceedings of the Western Joint Computer Conference*, pp. 295–298, 1959.

J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. In *Proceedings of International Conference on Learning Representations*, 2014.

C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *Neural Information Processing Systems (NIPS) Workshop*, 2016.

M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, 2016.

D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pp. 2224–2232, 2015.

F. A. Faber, L. Hutchison, , B. Huang, J. Gilmer, S. S. Schoenholz, G. E. Dahl, O. Vinyals, S. Kearnes, P. F. Riley, and O. A. von Lilienfeld. Prediction errors of molecular machine learning models lower than hybrid dft error. *J. Chem. Theory Comput.*, 13:5255 – 5264, 09 2017.

E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1960.

T. Gärtner, P. A. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. *Proceedings of the Annual Conference on Computational Learning Theory*, pp. 129–143, 2003.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

J. Hachmann, R. Olivares-Amaya, S. Atahan-Evrenk, C. Amador-Bedolla, R. S. Snchez-Carrera, A. Gold-Parker, L. Vogt, A. M. Brockway, and A. Aspuru-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*, 2011.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

T. Hofmann, B. Schölkopf, and A. J. Smola. Kernel methods in machine learning. *The Annals of Statistics*, 36:1171–1220, 2008.

P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:864–871, 1964.

T. S. Hy, S. Trivedi, H. Pan, B. M. Anderson, and R. Kondor. Predicting molecular properties with covariant compositional networks. *Journal of Chemical Physics*, 148, 2018.

W. Jin, C. W. Coley, R. Barzilay, and T. Jaakkola. Predicting organic reaction outcomes with weisfeiler-lehman network. *31st Conference on Neural Information Processing Systems (NIPS)*, 2017.

H. Kashima, K. Tsuda, and A. Inokuchi. Kernels for graphs. *Kernels and Bioinformatics*, pp. 155–170, 2004.

S. Kearns, K. McCloskey, M. Brendl, V. Pande, and P. Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of Computer-Aided Molecular Design*, 30:595–608, 2016.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proc. ICLR*, San Diego, 2015.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of International Conference on Learning Representations*, 2017.

W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:1133–1138, 1965.

R. Kondor, T. S. Hy, H. Pan, B. M. Anderson, and S. Trivedi. Covariant compositional networks for learning graphs. *https://arxiv.org/abs/1801.02144*, 2018.

N. M. Kriege, P. Giscard, and R. Wilson. On valid optimal assignment kernels and applications to graph classification. *Advances in Neural Information Processing Systems 29*, 2016.

C. Kyunghyun, V. M. Bart, G. Caglar, B. Dzmitry, B. Fethi, S. Holger, and B. Yoshua. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

M. Looks, M. Herreshoff, D. Hutchins, and P. Norvig. Deep learning with dynamic computation graphs. *International Conference of Learning Representations (ICLR)*, 2017.

L. V. D. Maaten and G. Hinton. Visualizing data using t-sne. *J. Mach Learn. Res.*, 9: 2579–2605, 2008.

D. J. C. Mackay. Gaussian processes: A replacement for supervised neural network? *Tutorial lecture notes for NIPS 1997*, 1997.

S. Mika, B. Schölkopf, A. Smola, K. B. Müller, M. Scholz, and G. Rätsch. Kernel pca and de-noising in feature spaces. *Advances in Neural Information Processing Systems 11*, 1998.

J. Munkres. Trie memory. *Journal of the Society for Industrial and Applied Mathematics*, 5:32–38, 1957.

M. Niepert, M. Ahmed, and K. Kutzkov. Learning convolutional neural networks for graphs. In *Proceedings of the International Conference on Machine Learning*, 2016.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. *Neural Information Processing Systems (NIPS)*, 2017.

R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. von Lilienfeld. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data*, 1, 2014.

S. Ravi and Q. Diao. Large scale distributed semi-supervised learning using streaming approximation. *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2016.

L. Ruddigkeit, R. van Deursen, L. C. Blum, and Jean-Louis Reymond. Enumeration of 166 billion organic small molecules in the chemical universe database gdb-17. *J. Chem. Inf. Model.*, 52:2864Äì2875, 2012.

K. T. Schütt, Kristof T., F. Arbabzadah, S. Chmiela, K. R. Müller, and A. Tkatchenko. Quantum-chemical insights from deep tensor neural networks. *Nature communications*, 2017.

N. Shervashidze, S. V. N. Vishwanathan, T. H. Petri, K. Mehlhorn, and K. M. Borgwardt. Efficient graphlet kernels for large graph comparison. *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2009.

N. Shervashidze, P. Schweitzer, E. J. van Leeuwan, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

S. V. N. Vishwanathan. Kernel methods: Fast algorithms and real life applications. *PhD thesis, Indian Institute of Science, Bangalore, India*, 2002.

S. V. N. Vishwanathan, N. N. Schraudolf, R. Kondor, and K. M. Bogwardt. Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242, 2010.

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