Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Geometric Deep Learning

Author:
Shunwang Gong

Supervisor:
Stefanos Zafeiriou

Submitted in partial fulfillment of the requirements for the MSc degree in Advanced Computing of Imperial College London

September 2018

Abstract

Machine Learning on graphs and manifolds are important ubiquitous tasks with applications ranging from network analysis to 3D shape analysis. Traditionally, machine learning approaches relied on user-defined heuristics to extract features encoding structural information about a graph or mesh. Recently, there has been an increasing interest in *geometric deep learning* [6] that automatically learns signals defined on graphs and manifolds. We are then motivated to apply such methods to address the multifaceted challenges arising in computational biology and computer graphics for decades, *i.e.* protein function prediction and 3D facial expression recognition. Here we propose a deep graph neural network to successfully address the semi-supervised multi-label classification problem (*i.e.* protein function prediction). With regard to 3D facial expression recognition, we propose a deep residual B-Spline graph convolution network, which allows for end-to-end training and inference without using hand-crafted feature descriptors. Our method outperforms the current baseline results on 4DFAB [10] dataset.

Acknowledgments

I would like to thank my supervisor Dr Stefanos Zafeiriou, and Prof Michael Bronstein for all their support and guidance throughout the thesis.

Additionally I'd like to thank all of the researchers for providing support, resources, and conversations, especially Dr Vladimir Gligorijevic (Simons Foundation, my collaborator on the problem of *Protein Function Prediction*), Shiyang Cheng (PhD student at iBug group Imperial College London, providing me valuable 4DFAB database [10] (2017) to start my work on analyzing *3D Facial Expressions*), Mehdi Bahri (PhD student at Imperial College London), Matthias Fey (PhD student at TU Dortmund University), Yun Wang (PhD student at Language Technologies Institute (LTI), Carnegie Mellon University (CMU)).

Finally, I'd like to thank the Computing Support Group at Imperial College London for providing me with a NVIDIA GTX 1080 and GPU clusters used in this thesis.

Contents

Chapter 1

Introduction

1.1 Overview

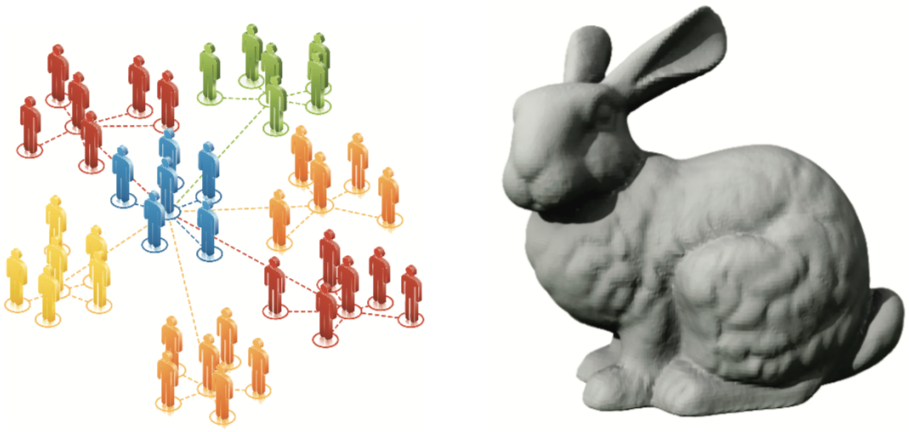
1.1.1 Traditional Machine Learning on Graphs and Manifolds

Significance *Graphs* and *manifolds* (Fig. 1.1) are ubiquitous data structures, employed extensively within computer science and related fields. Social networks, molecular graph structures, biological protein-protein networks, recommender systems, 3D computer vision, all of these domains and many more can be readily formulated as graphs or manifolds, which capture interactions (*i.e.* edges) between individual units (*i.e.* nodes). As a consequence of their ubiquity, graphs are the backbone of countless systems, allowing relational knowledge about interacting entities to be efficiently stored and accessed [2].

Many machine learning applications seek to make predictions or discover new patterns using graph-structured data as feature information. For example, one might wish to classify the role of a protein in biological interaction networks, predict the role of a person in a collaboration network, recommend new friends to a user in a social network, or predict new therapeutic applications of existing drug molecules, whose structure can be represented as a graph.

Challenges The central problem in machine learning on graphs is finding a way to incorporate information about graph structure into a machine learning model. For example, in the case of link prediction in a social network, one might want to encode pairwise properties between nodes, such as relationship strength or the number of common friends. Or in the case of node classification, one might want to cluster nodes from the information defined on top of each node and the graph topologocial structure (See fig.). The challenge from a machine learning perspective is that there is no straightforward way to encode this high-dimensional, non-Euclidean information into a feature vector.

Traditional Methods To extract structural information from graphs, traditional machine approaches often rely on summary graph statistics (*e.g.* degrees or clustering coefficients) [3], kernel functions [62], or carefully engineered features to measure local neighborhood structures [39]. However, these approaches are limited because these hand-engineered



(a) Graphs (b) Manifolds

Figure 1.1: The underlying data structure of a social network or 3D shape is represented as graphs (left) or manifolds (right).

features are inflexible (*i.e.* they cannot adapt during the learning process) and designing these features can be a time-consuming and expensive process. This then proposed the requirement of the highly-efficient representation learning methods on non-Euclidean data.

1.1.2 Widespread Prosperity of Deep Learning on Euclidean Domains

Representation Learning Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification. Deep learning methods are representation learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. The key aspect of deep learning is that these layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure.

Deep Learning Models Since the first time a convolutional network won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), one of the largest contest in object recognition in 2012, deep learning models have gained prosperity and proven extremely successful on a wide variety of tasks, from computer vision and acoustic modeling to natural language processing [37]. One of the key reasons for the success of deep neural networks is their ability to leverage statistical properties of the data such as stationarity and compositionality through local statistics, which are present in natural images, video, and speech. These properties are exploited efficiently by Convolutional Neural Networks (CNNs) [38]. Precisely, CNNs extract the local stationarity property of the input data or signals by revealing local features that are shared across the data domain. These similar features are identified with localized convolutional filters or kernels, which are learned

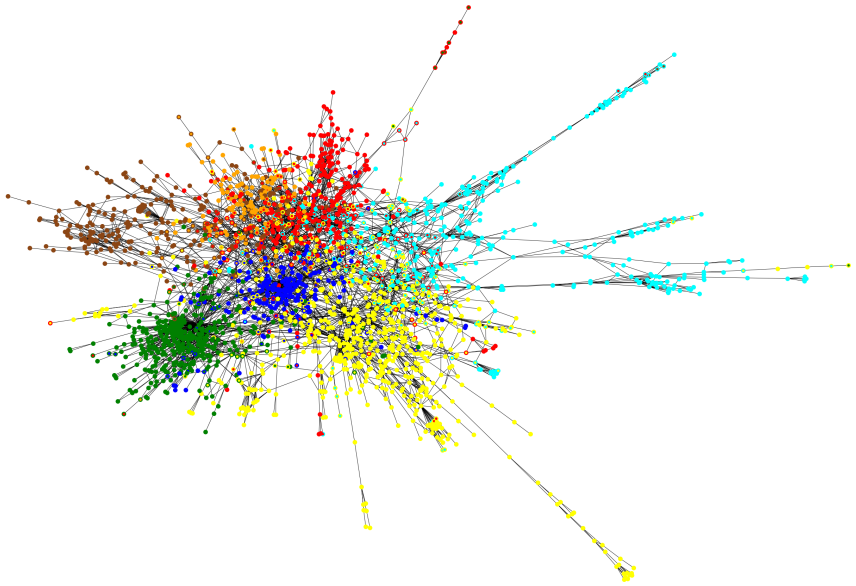


Figure 1.2: Classifying research papers in the CORA dataset with GCN [34]. Shown is the citation graph, where each node is a paper, and an edge represents a citation. Vertex fill and outline colors represents the predicted and groundtruth labels, respectively; ideally, the two colors should coincide. We produce the figure with GCN [34])

from the data.

While deep learning has been proven to be powerful tools in various fields, it is not straightforward to apply CNNs to graphs as basic operators like convolution and pooling are not well defined on graphs. This makes such extension challenging, both theoretically and implementation-wise.

1.1.3 Geometric Deep Learning

The term of *geometric deep learning* is first proposed by Bronstein et al. [6] for emerging techniques attempting to generalize deep neural models to non-Euclidean domains such as graphs and manifolds. The interest in this field has exploded in the past years, resulting in numerous successful attempts to apply these methods in a broad spectrum of problems ranging from biochemistry [17] to recommender systems [44]. In this thesis, we propose to solve two fundamental problems lied in computational biology and 3D computer vision with geometric deep learning.

1.2 Background

In this thesis, we tackle two fundamental problems lied in computational biology and 3D computer vision communities: protein function prediction and 3D facial expression analysis (*i.e.* recognition). For illustrated in Figure 1.3, two problems are closely connected to geometric deep learning. Next, we give a briefly overview of each problem and highlight the corresponding challenges.

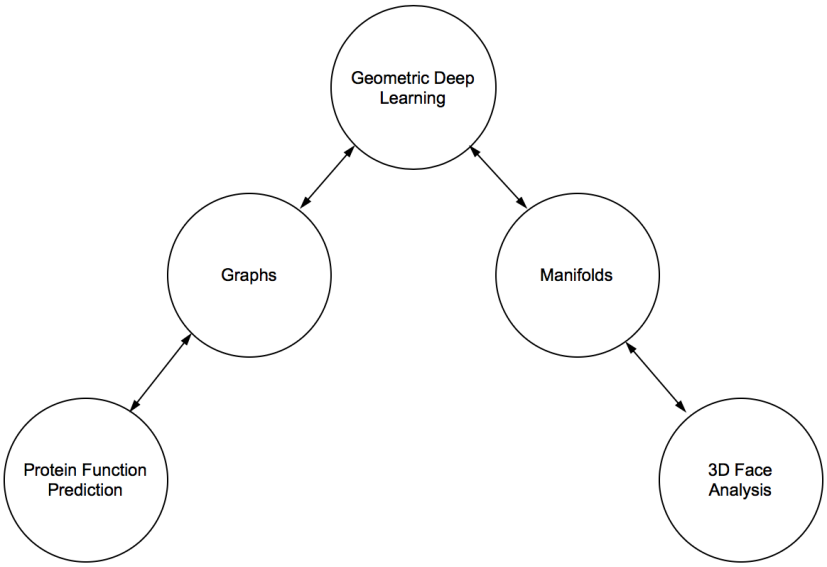


Figure 1.3: Two problems towards geometric deep learning.

1.2.1 Protein Function Prediction

Over the past a few years, an abundance of large-scale protein interaction networks (see Figure 1.4) was produced by high-throughput experimental methods. These networks serve as a representation of the evidence for similar function within a group of proteins, where nodes represent proteins and are linked to each other by edges representing evidence of shared function. Developing methods for protein function prediction allows us to maximize the utility of functional annotations derived from costly and time consuming protein function characterization and large-scale genomics experiments.

Today, a common challenge for network-based methods is how to capture underlying feature representations from complex and highly non-linear topological structures. *GeneMANIA* (Mostafavi et al. [46], Mostafavi and Morris [45] (2008, 2012)) is a widely used semi-supervised network-based method that first integrates kernels of different network types into a single kernel by solving a constrained linear regression problem; then, it applies Gaussian label propagation on the resulting kernel to make label predictions. However, as pointed out by Cho et al. [11], this method suffer from the information loss incurred when combining all the network types into a single network. Recently, Cho et al. [11] (2016) proposed *Mashup*, a network integration framework, to address the challenge of fusing noisy and incomplete interaction networks. *Mashup* takes as input a collection

of protein interaction networks and applies a matrix factorization-based technique to construct compact low-dimensional vector representation of proteins that best explains their wiring patterns across all networks. These vectors are then fed into a *support vector machine* (SVM) classifier to predict functional labels of proteins. The key step in *Mashup* is the feature learning step that constructs informative features that have been shown to be useful in multiple scenarios including highly accurate protein function prediction. *deepNF* (Gligorijević et al. [19] (2017)) is the first method based on a *multimodal deep autoencoder* (MDA) to extract compact, low-dimensional feature representations from traning MDA with different heterogeneous protein interaction networks. The learned feature is then used to train SVM classifer to perform the task of multilable semi-supervised classification. They argue that features learned by using MDA lead to substantial improvements in protein function prediction accuracy. *deepNF* outperforms the previous benchmarks from *GeneMANIA* and *Mashup* on the dataset STRING human and yeast in terms of molecular function (MF), biological process (BP), cellular component (CC) GO terms.

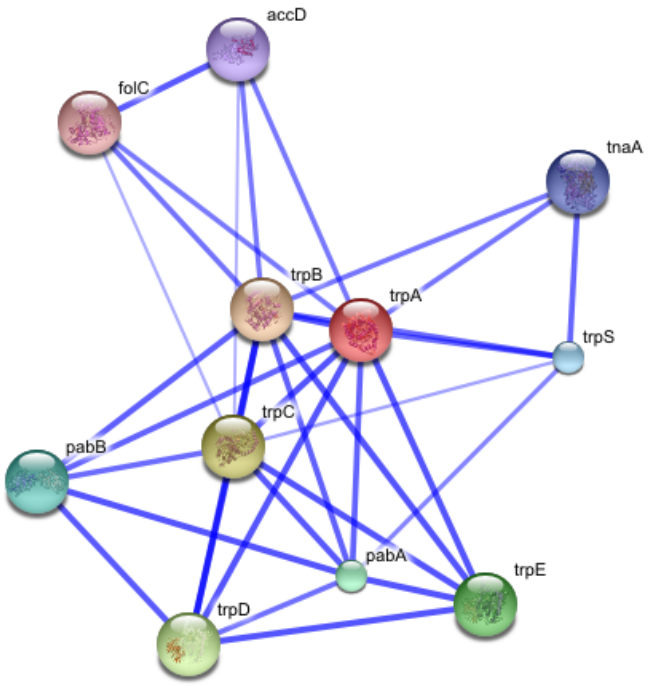
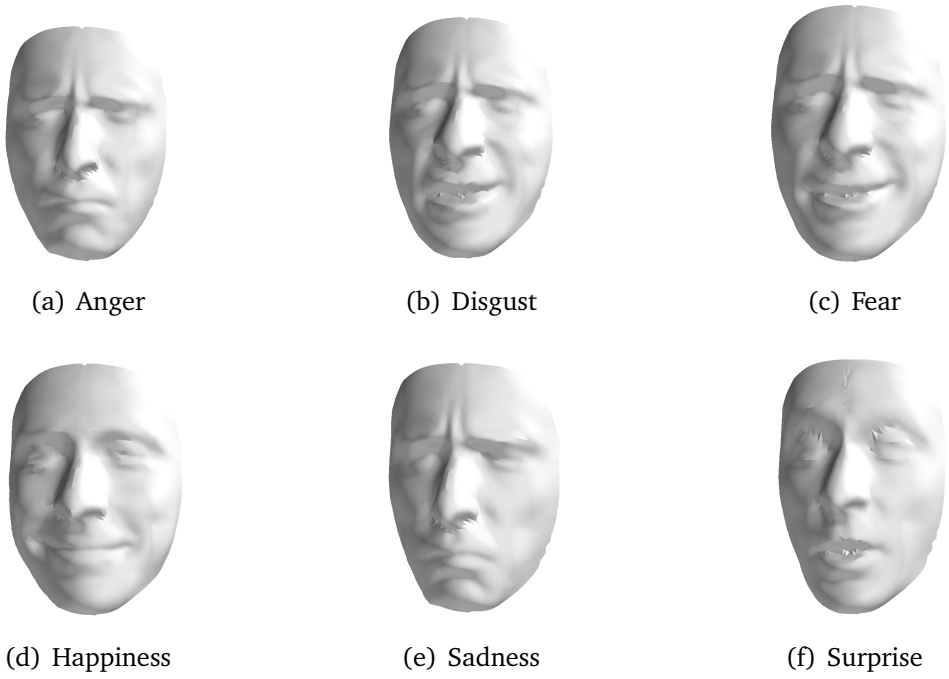


Figure 1.4: An example protein interaction network, produced through the STRING web resource. Patterns of protein interactions within networks are used to infer function. Here, products of the bacterial *trp* genes coding for *tryptophan synthase* are shown to interact with themselves and other, related proteins.

1.2.2 3D Facial Expression Recognition

Facial expression is the most cogent, naturally preeminent means for humans to communicate emotions, to clarify and give emphasis, to express intentions and, more generally,



(a) Anger (b) Disgust (c) Fear

(d) Happiness (e) Sadness (f) Surprise

Figure 1.5: Examples of 6 three-dimensional facial expressions of one participant in the 4DFAB database.

to regulate interactions with the environment and other people [1]. These facts highlight the importance of automatic facial behaviour analysis, including facial expression of emotion and facial action unit (AU) recognition. It can be regarded as the essence of next-generation computing systems as it plays a crucial role in affective computing technologies (*i.e.* proactive and affective user interfaces), learner-adaptive tutoring systems, patient-profiled personal well-being technologies, etc [48].

In the past, the majority of work conducted in this area involves 2D images, despite the problems this presents due to inherent pose and illumination variations. In order to deal with these problems, 3D face are increasingly used in expression anaylsis because all the data is in the form of coordinates, containing much more information than a flat image, but the progress in this field was restrained by the lack of high resolution 3D facial expression database.

Recently, the iBug group in Imperial College London published high-resolution 3D face dataset 4DFAB (Cheng et al. [10] (2017)), which includes 180 participants on 4 different recording sessions and each participants performs 4 to 6 basic expressions (*i.e.* anger, disgust, fear, happiness, sadness and surprise) (see Figure 1.5). They reported the current baseline performance of 3D facial expression recognition on 4DFAB database with a recognition rate of 70.27% Session 1, 69.02%, 66.91% and 68.89% in Session 2, 3 and 4 experiments respectively. The method they proposed requires to first extracting the main face regions (covering eyes, mouth, cheeks and nose) based on 79 facial landmarks. The region was further divided into non-overlapping blocks, for which *Histogram of Oriented Normal Vectors* (HONV) [60] were computed. After this, PCA and LDA were used for dimensionality reduction, a multi-class SVM [9] was employed to classify expressions.

1.3 Thesis Statement

This thesis seeks to address the multifaceted challenges arising in computational biology (*e.g.* protein function prediction) and 3D facial expression analysis (*e.g.* recognition). In particular, this thesis provides evidence to support the following statement:

Thesis Statement: *With appropriate architectures and algorithm design, geometric deep learning methods are capable of providing general solutions to graph- and mesh- structured data in terms of various tasks, e.g. multi-label/multi-class semi-supervised node classification, 3d shape recognition.*

We believe that the architectures and the methods developed in this thesis will enable more people to take advantage of the power of geometric deep learning to resolve problems in different fields where the underlying data structure is graphs or manifolds. All the codes developed when completing this thesis will be made publicity available.

1.4 Contributions

Our major contributions in this thesis are summarised below:

- We present a review of the most recent spectral and spatial graph convolution methods with detailed analysis of designing intuitions behind those methods. Additionally, we evaluate and compare those methods on two popular tasks, node feature classification and shape correspondence.
- We propose a graph autoencoder network incorporated with SVM classifier. The architecture is capable of learning a compact, low-dimensional feature representation in an unsupervised way. The extracted latent featuer can be used to train a SVM classifier to solve semi-supervised multi-label classification problem.
- We introduce a novel deep graph neural network, which allows to learn graph topological structure without providing any hand-crafted node feature. Our model achieved the state-of-the-art performance on BIOGRID human, mouse and yeast dataset. We argue that our method can be applied to any task if the underlying data can be constructed into a graph structure.
- We provide a fast method based on continuous B-spline kernels on the problem of 3D facial expression recognition, which allows for end-to-end training without using hand-crafted feature descriptors. Our method outperforms the current baseline results on 4DFAB dataset.
- All the codes developed when completing this thesis will be made publicly available. Currently, we provide the GPU implementation based on python3 on top of Pytorch, a deep learning framework.

Chapter 2

Preliminaries on Deep Learning from Euclidean Domains

In this chapter we provide some background information of deep learning that are the basic knowledge to understand our models developed on non-Euclidean domains. We start with giving a brief review of *Convolutional Neural Networks* (CNNs), including convolutional, pooling, activation and fully connected layer. We then discuss the *Autoencoder* (AE).

2.1 Convolutional Neural Networks

A CNN architecture is formed by a stack of distinct layers that transform the input volume into an output volume (*e.g.* holding the class scores) through a differentiable function (Figure (2.1)). A few distinct types of layers (*i.e.* convolution, pooling, fully connected layer) are commonly used. We discuss them further below:

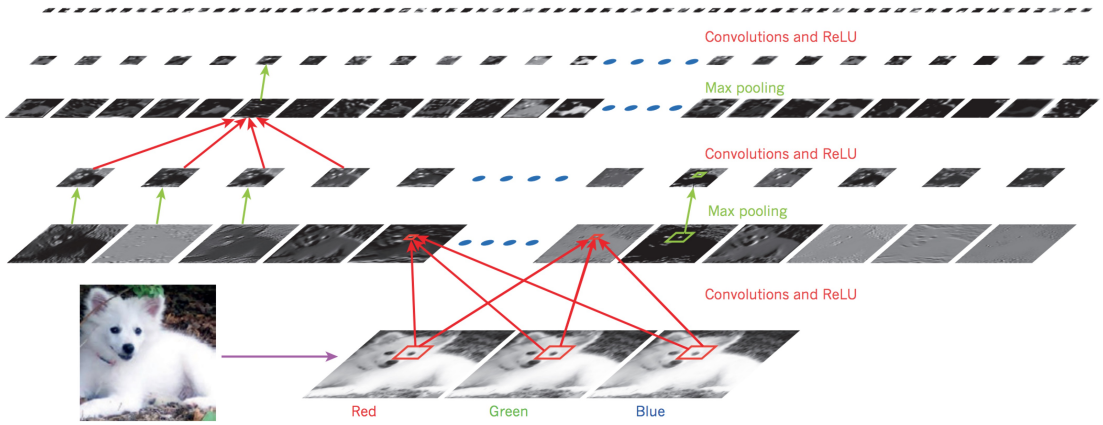


Figure 2.1: Typical convolutional neural network architecture [37] used in computer vision applications.

2.1.1 Convolutional Layer

Overview and Intuition

The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable **filters** (or kernels), which have a small **receptive field** (see Figure 2.2), but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input.

Stacking the **activation maps** for all filters along the depth dimension forms the full output volume of the convolution layer. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at a small region in the input and shares parameters with neurons in the same activation map.

For each convolutional layer of the form $\mathbf{g} = \mathcal{C}_\Gamma(\mathbf{f})$, acting on a p-dimensional input $\mathbf{f}(x) = (f_1(x), \ldots, f_p(x))$ by applying a bank of filters $\Gamma = (\gamma_{l,l'})$, $l = 1, \ldots, p, l' = 1, \ldots, q$ and point-wise non-linear activation function ξ,

$$g_{l'}(x) = \xi(\sum_{l=1}^{p}(f_l * \gamma_{l,l'})(x)) \tag{2.1}$$

producing a q-dimensional output $\mathbf{g}(x) = (g_1(x), \ldots, g_q(x))$ often referred to as the *feature maps*. For compact support filters, the space complexity of parameters in per filter is $\mathcal{O}(1)$ (independent of input image size n) and the computational complexity is $\mathcal{O}(n)$.

Local connectivity

When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume because such a network architecture does not take the spatial structure of the data into account. Convolutional networks exploit **spatially local correlation** by enforcing a **local connectivity pattern** between neurons of adjacent layers: each neuron is connected to only a small region of the input volume. The extent of this connectivity is a hyperparameter called the *receptive field* of the neuron (see Figure 2.2). The connections are local in space (along width and height), but always extend along the entire depth of the input volume. Such an architecture ensures that the learnt filters produce the strongest response to a **spatially local input pattern**.

Parameter sharing

A parameter sharing scheme is used in convolutional layers to **control the number of free parameters**. It relies on one reasonable assumption: That if a patch feature is useful to compute at some spatial position, then it should also be useful to compute at other positions. In other words, denoting a single 2-dimensional slice of depth as a depth slice, we constrain the neurons in each depth slice to use the same weights and bias.

Since all neurons in a single depth slice share the same parameters, then the forward pass in each depth slice of the convolutional layer can be computed as a convolution of

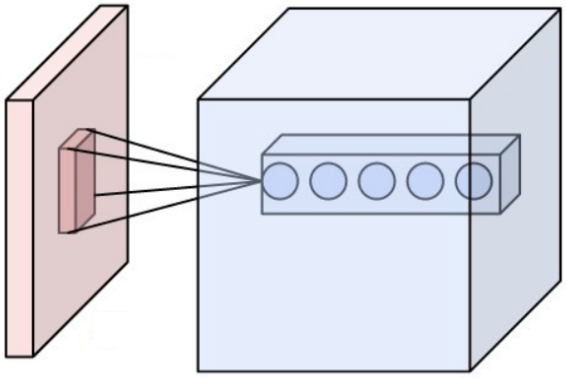


Figure 2.2: Neurons of a convolutional layer (blue), connected to their receptive field (red)

the neuron's weights with the input volume. Therefore, it is common to refer to the sets of weights as a filter (or a kernel), which is convolved with the input. The result of this convolution is an activation map, and the set of activation maps for each different filter are stacked together along the depth dimension to produce the output volume. Parameter sharing contributes to the **translation invariance** of the CNN architecture. Thanks to this property, the linear operator at each layer have a **constant number of parameters**, independent of the input size n.

2.1.2 Pooling Layer

Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which *max pooling* is the most common. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. The intuition is that the exact location of a feature is less important than its rough location relative to other features. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also **control overfitting**. It is common to periodically insert a pooling layer between successive convolutional layers in a CNN architecture. The pooling operation provides another form of **translation invariance**.

The pooling layer operates independently on every depth slice of the input and resizes it spatially. The most common form (see Figure 2.3) is a pooling layer with filters of size 2×2 applied with a stride of 2 downsamples at every depth slice in the input by 2 along both width and height, discarding 75% of the activations. In this case, every max operation is over 4 numbers. The depth dimension remains unchanged.

General Pooling

Generally, a *downsampling/subsampling* or *pooling layer* can be defined as

$$g_{l'}(x) = P(\{f_l(x') : x' \in \mathcal{N}(x)\}), \quad l = 1, \dots, p \tag{2.2}$$

where $\mathcal{N}(x) \subset \mathcal{V}$ is a neighborhood around x and P is a *permutation-invariant function* such as a L_p-norm (*i.e.* the choice of $p = 1, 2$ or ∞ results in *average-*, *energy-*, or *max-pooling*). *Average pooling* was often used historically but has recently fallen out of favor compared to the *max pooling* operation, which has been shown to work better in practice [52].

Due to the aggressive reduction in the size of the representation, the trend is towards using smaller filters [22] or discarding the pooling layer altogether [58].

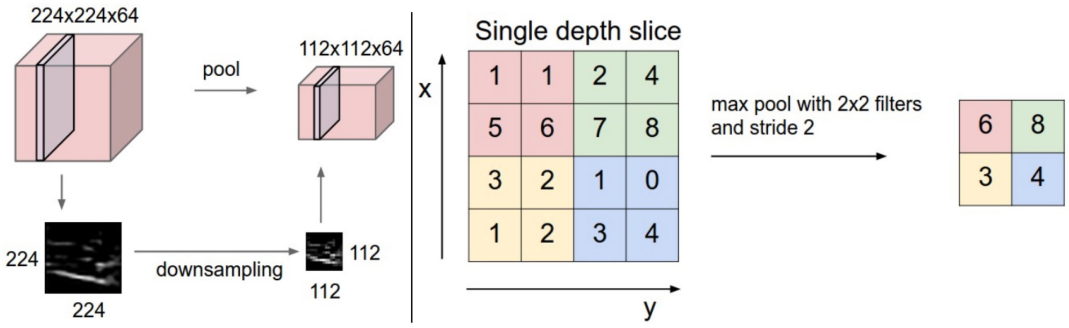


Figure 2.3: Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume. Left: In this example, the input volume of size [224x224x64] is pooled with filter size 2, stride 2 into output volume of size [112x112x64]. Notice that the volume depth is preserved. Right: The most common downsampling operation is max, giving rise to max pooling, here shown with a stride of 2. That is, each max is taken over 4 numbers (little 2x2 square).

2.1.3 Activation Layer

Here we mainly introduce the activation functions we used in our models, *i.e.* ReLU, Leaky ReLU, PReLU, ELU, Sigmoid, Tanh activation function.

ReLU (Figure 2.4(a)) is the abbreviation of *Rectified Linear Units*. This layer applies the *non-saturating activation function*. It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer.

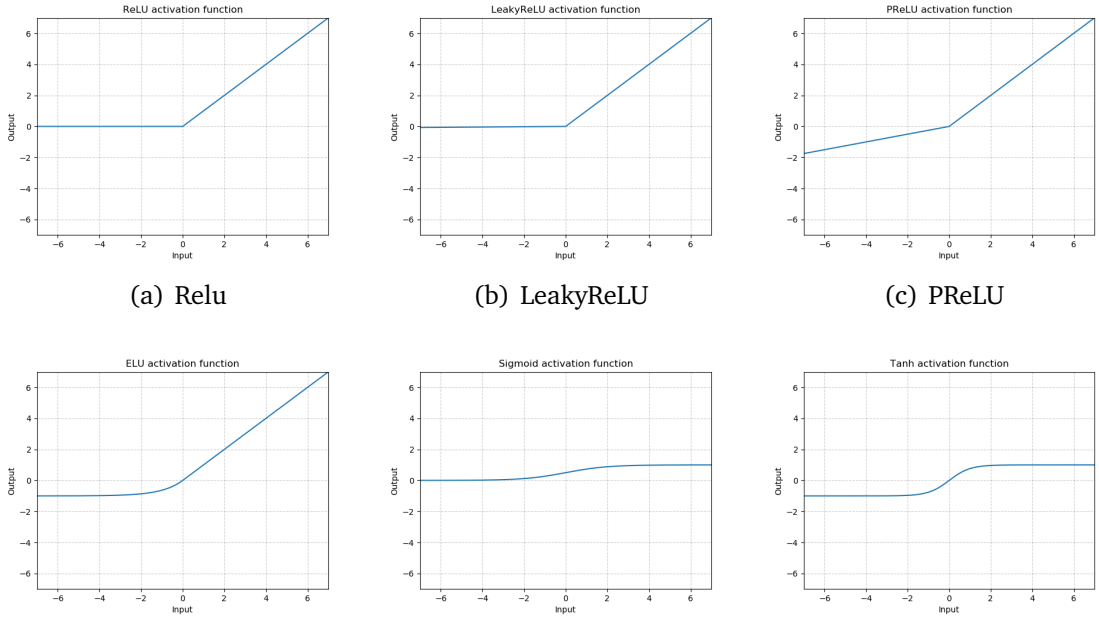
$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.3}$$

LeakyReLU (Figure 2.4(b)) allows a small, positive gradient when the unit is not active.

$$\mathrm{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases} \tag{2.4}$$

Parametric ReLU (PReLU) (Figure 2.4(c)) takes this idea further by making the coefficient of leakage into a parameter that is learned along with the other neural network parameters [24].

$$\mathrm{PReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases} \tag{2.5}$$



(a) Relu (b) LeakyReLU (c) PReLU

(d) ELU (e) Sigmoid (f) Tanh

Figure 2.4: Examples of 6 activation functions we used in the thesis.

Exponential linear unit (ELU) (Figure 2.4(d)) tries to make the mean activations closer to zero which speeds up learning. It has been shown that ELUs can obtain higher classification accuracy than ReLUs [13].

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ a(e^x - 1) & \text{otherwise} \end{cases} \tag{2.6}$$

a is a hyper-parameter to be tuned and $a \geq 0$ is a constraint.

Sigmoid activation function (Figure 2.4(e)) refers to the special case of the logistic function shown in the Figure and defined by the equation (2.7). Sigmoid functions have domain of all real numbers, with return value monotonically increasing most often from 0 to 1. This is the property we take use of in some cases where we need the output in this domain.

$$\mathrm{Sigmoid}(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{2.7}$$

Hyperbolic tangent (Tanh) (Figure 2.4(f)) is also used sometimes, defined as:

$$\mathrm{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.8}$$

2.1.4 Fully Connected Layer

Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have

connections to all activations in the previous layer, as seen in regular neural networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

2.2 Autoencoder

Typically, the aim of an autoencoder is to learn a representation (encoding) for a set of data, for the purpose of **dimensionality reduction**. *deepNF* used this property to extract the compact, low dimensional feature representation of heterogeneous protein interaction networks. An alternative use is as a **generative model**: for example, if a system is manually fed the codes it has learned for "cat" and "flying", it may attempt to generate an image of a flying cat, even if it has never seen a flying cat before [33].

2.2.1 Structure

An autoencoder always consists of two parts, the encoder and the decoder, which can be defined as transitions ϕ and ψ. In the most of cases, the objective function is defined as $L2$ Loss, such that:

$$\begin{aligned} &\phi : \mathcal{X} \to \mathcal{F} \\ &\psi : \mathcal{F} \to \mathcal{X} \\ &\phi, \psi = \arg\min \|X - (\psi \circ \phi)X\|^2 \end{aligned} \tag{2.9}$$

where X represents the input.

In the simplest case (Figure 2.5), where there is one hidden layer, the encoder stage of an autoencoder takes the input $\mathbf{x} \in \mathbb{R}^d = \mathcal{X}$ and maps it to $\mathbf{z} \in \mathbb{R}^p = \mathcal{F}$:

$$\mathbf{z} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.10}$$

The image $\mathbf{z}$ is usually referred to as latent representation. Here, σ is an element-wise activation function such as a sigmoid function (Figure 2.4(e)) or a ReLU (Figure 2.4(a)). $\mathbf{W}$ is a weight matrix and $\mathbf{b}$ is a bias vector. After that, the decoder stage of the autoencoder maps $\mathbf{z}$ to the reconstruction $\mathbf{x}'$ of the same shape as $\mathbf{x}$:

$$\mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{z} + \mathbf{b}') \tag{2.11}$$

where $\sigma', \mathbf{W}'$ and $\mathbf{b}'$ for the decoder may differ in general from the corresponding $\sigma, \mathbf{W}$, and $\mathbf{b}$ for the encoder, depending on the design of the autoencoder.

Autoencoders are also trained to minimise reconstruction errors (such as squared errors):

$$\mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2 \tag{2.12}$$

where $\mathbf{x}$ is usually averaged over some input training set.

If the feature space $\mathcal{F}$ has lower dimensionality than the input space $\mathcal{X}$, then the feature vector $\phi(\mathbf{x})$ can be regarded as a compressed representation of the input $\mathbf{x}$.

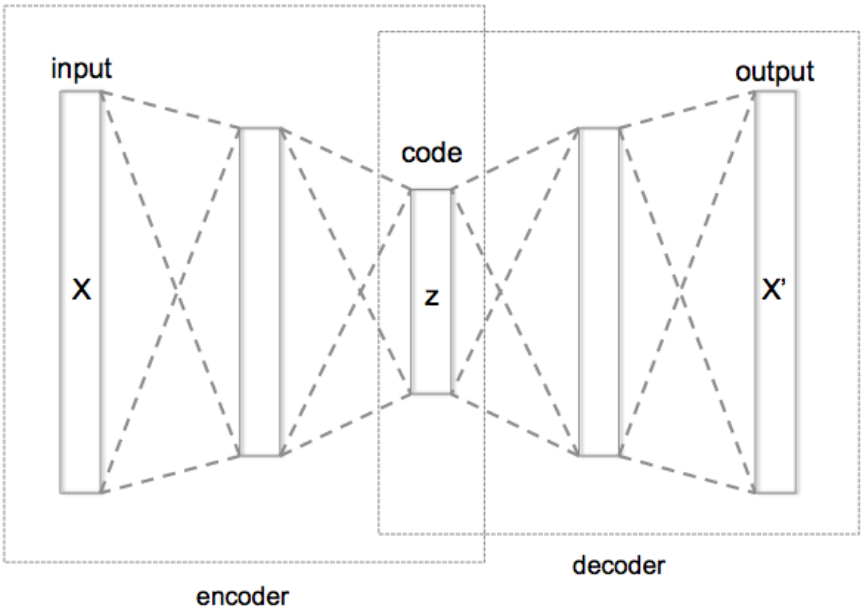


Figure 2.5: The structure of a standard autoencoder.

2.2.2 Denoising Autoencoder

Denoising autoencoder is a variation of standard autoencoders. Since the underlying architecture of deepNF is based on denoising autoencoder, we give a simple introduction of this structure.

Basically, denoising autoencoders take a partially corrupted input whilst training to recover the original undistorted input. This technique has been introduced with a specific approach to good representation [61]. A good representation is one that can be obtained robustly from a corrupted input and that will be useful for recovering the corresponding clean input. This definition contains the following implicit assumptions:

- The higher level representations are relatively stable and robust to the corruption of the input.
- It is necessary to extract features that are useful for representation of the input distribution.

To train an autoencoder to denoise data, it is necessary to perform preliminary stochastic mapping $\mathbf{x} \rightarrow \tilde{\mathbf{x}}$ in order to corrupt the data and use $\tilde{\mathbf{x}}$ as input for a normal autoencoder, with the only exception being that the loss should be still computed for the initial input $\mathcal{L}(\mathbf{x}, \tilde{\mathbf{x}}')$ instead of $\mathcal{L}(\tilde{\mathbf{x}}, \tilde{\mathbf{x}}')$.

Chapter 3

Preliminaries on Graph Theory

In this chapter, we review some basic definitions and notations from spectral graph theory, with a focus on how it enables us to extend many of the important mathematical ideas and intuitions from *classical Fourier Analysis* to the graph setting, which is considered as the foundation to understand the designing intuition of *spectral graph convolution* we would introduce in the next chapter. We also introduce graph coarsening theory, a key ingredient of graph subsampling or pooling operation. This makes it possible for the model to learn hierarchical 3D shape representations and also accelerate the training process.

3.1 Basic Defintions and Notations

In this thesis, We are interested in analyzing signals defined on an undirected, connected, weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$, which consists of a finite set of vertices $\mathcal{V}$ with $|\mathcal{V}| = n$, a set of edges $\mathcal{E}$, and a weighted adjacency matrix $\mathbf{W}$. If there is an edge $e = (i, j)$ connecting vertices i and j, the entry W_{ij} (or a_{ij}) represents the weight of the edge; otherwise, $W_{ij} = 0$. In some cases if only provided the adjacency matrix $\mathcal{A}$, the entry $W_{ij} = 1$ if vertex i and j are connected; otherwise, $W_{ij} = 0$.

When the edge weights are not naturally defined by an application, one common way to define the weight of an edge connecting vertices i and j is via a *thresholded Gaussian kernel* weighting function:

$$W_{ij} = \begin{cases} \exp(-\frac{[dist(i,j)]^2}{2\theta^2}) & \text{if } dist(i,j) \leq \mathcal{K} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

for some parameters θ and $\mathcal{K}$. In (3.1), $dist(i, j)$ may represent a physical distance between vertices i and j, or the Euclidean distance between two feature vectors describing i and j, the latter of which is especially common in graph-based semi-supervised learning methods. A second common method is to connect each vertex to its k-nearest neighbors based on the physical or feature space distances. For other graph construction methods, see, e.g., [21].

A *signal* or *vertex function* $f : \mathcal{V} \to \mathbb{R}$ defined on the vertices of the graph may be represented as a vector $\mathbf{f} \in \mathbb{R}^N$, where the i^{th} component of the vector $\mathbf{f}$ represents the function value at the i^{th} vertex in $\mathcal{V}$. The *graph signal* in Figure 3.1 is one example.

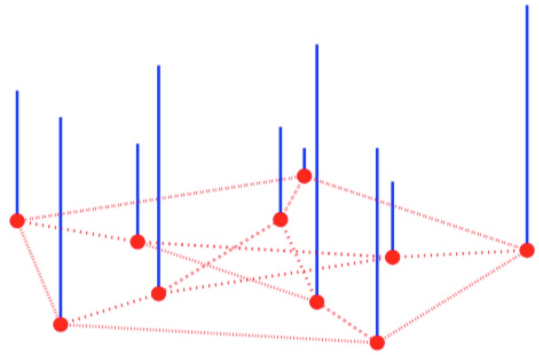


Figure 3.1: A random positive graph signal on the vertices of the Petersen graph. The height of each blue bar represents the signal value at the vertex where the bar originates.

3.2 Spectral Graph Theory

3.2.1 Graph Laplacian

Definition 3.2.1 (non-normalized graph Laplacian)**.** The *non-normalized graph Laplacian*, also called the *combinatorial graph Laplacian*, is defined as $\mathbf{\Delta} = \mathbf{D} - \mathbf{W}$, where the *degree matrix* $\mathbf{D} = \mathrm{diag}(\sum_j W_{ij})$. The graph Laplacian is a difference operator, as, for any signal $\mathbf{f} \in \mathbb{R}^N$, it satisfies

$$(\mathbf{\Delta} f)(i) = \sum_{j \in \mathcal{N}_i} W_{ij}[f(i) - f(j)],$$

where the neighborhood $\mathcal{N}_i$ is the set of vertices connected to vertex i by an edge. More generally, we denote by $\mathcal{N}(i,k)$ the set of vertices connected to vertex i by a path of k or fewer edges.

Because the graph Laplacian $\mathbf{\Delta}$ is a real symmetric matrix, it has a complete set of orthogonal eigenvecotrs, which we denote by $\mathbf{\Phi} = (\phi_1, \ldots, \phi_n)$[1]. These eigenvectors have associated real, non-negative eigenvalues $\lambda_1, \ldots, \lambda_n$, satisfying $\mathbf{\Delta}\phi_i = \lambda_i \phi_i$, for $i = 1, 2, \ldots, N$. Zero appears as an eigenvalue with multiplicity equal to the number of connected components of the graph [12], and thus, since we consider connected graphs, we assume the graph Laplacian eigenvalues are ordered as $0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_N := \lambda_{max}$.

Definition 3.2.2 (normalized graph Laplacian)**.** By normalizing each weight W_{ij} by a factor of $\frac{1}{\sqrt{d_i d_j}}$, where d_i represents *the degree of vertex* i (*i.e.* $d_i = \sum_j W_{ij}$), we then get the *normalized graph Laplacian* defined as $\widetilde{\mathbf{\Delta}} = \mathbf{D}^{-1/2}\mathbf{\Delta}\mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}$.

The eigenvalues $\tilde{\lambda}_1, \ldots, \tilde{\lambda}_N$ of the normalized graph Laplacian of a connected graph $\mathcal{G}$ satisfy

$$0 \leq \tilde{\lambda}_1 \leq \tilde{\lambda}_2 \leq \ldots \leq \tilde{\lambda}_{max} \leq 2,$$

[1]Note that there is not necessarily a unique set of graph Laplacian eigenvectors, but we assume throughout that a set of eigenvectors is chosen and fixed.

with $\tilde{\lambda}_{max} = 2$ if and only if $\mathcal{G}$ is *bipartite*; i.e. the set of vertices $\mathcal{V}$ can be partitioned into two subsets $\mathcal{V}_1$ and $\mathcal{V}_2$ such that every edge $e \in \mathcal{E}$ connects one vertex in $\mathcal{V}_1$ and one vertex in $\mathcal{V}_2$. We denote the normalized graph Laplacian eigenvectors by $\widetilde{\mathbf{\Phi}} = (\tilde{\phi}_1, \ldots, \tilde{\phi}_N)$.

As discussed in detail in the next section, both the normalized and non-normalized graph Laplacian eigenvectors can be used as filtering basis. However, from the current trend, normalized graph Laplacian is more widely used as it has the nice property that its spectrum is always contained in the interval $[0, 2]$ and, for bipartite graphs, the spectral folding phenomenon [47] can be exploited.

Definition 3.2.3 (Eigendecomposition of graph Laplacian)**.** Since both normalized and non-normalized Laplacian are symmetric and positive semi-definite matrices, they admit an eigendecomposition $\mathbf{\Delta} = \mathbf{\Phi}\mathbf{\Lambda}\mathbf{\Phi}^\mathsf{T}$, where the diagonal matrix $\mathbf{\Lambda} = \mathrm{diag}(\lambda_1, \ldots, \lambda_N)$.

3.2.2 Graph Fourier Transform

Definition 3.2.4 (classical Fourier transform)**.** The classical Fourier transform

$$\hat{f}(\xi) := < f, e^{2\pi i \xi t} > = \int_{\mathbb{R}} f(t) e^{-2\pi i \xi t} dt$$

is the expansion of a function f in terms of the complex exponentials, which are the *eigenfunctions* of the one-dimensional Laplace operator:

$$-\Delta(e^{2\pi i \xi t}) = -\frac{\partial^2}{\partial t^2}(e^{2\pi i \xi t}) = (2\pi\xi)^2 (e^{2\pi i \xi t}) \tag{3.2}$$

Definition 3.2.5 (graph Fourier transform)**.** we can define the *graph Fourier transform* $\hat{f}$ of any function $\mathbf{f} \in \mathbb{R}$ on the vertices of $\mathcal{G}$ as the expansion of $\mathbf{f}$ in terms of the eigenvectors of the graph Laplacian:

$$\hat{f}(\lambda_l) := < \mathbf{f}, \phi_l > = \sum_{i=1}^{N} f(i) \phi_l^*(i) \tag{3.3}$$

In a matrix-vector notation,

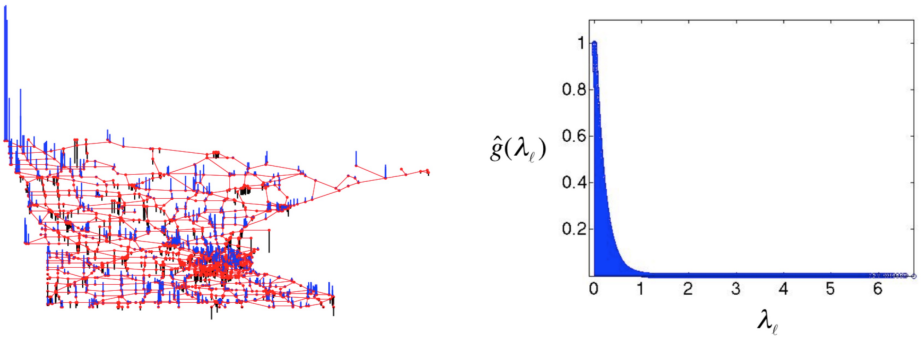
$$\hat{\mathbf{f}} = \mathbf{\Phi}^\mathsf{T} \mathbf{f} \tag{3.4}$$

Definition 3.2.6 (interse graph Fourier transform)**.** The *inverse graph Fourier transform* is then given by

$$f(i) = \sum_{l=1}^{N} \hat{f}(\lambda_i) \phi_l(i) \tag{3.5}$$

and the corresponding matrix-vector notation,

$$\mathbf{f} = \mathbf{\Phi}\, \hat{\mathbf{f}} \tag{3.6}$$



(a) spatial domain (b) spectral domain

Figure 3.2: Equivalent representations of a graph in the spatial and the spectral domains. In this case, the signal is a *heat kernel* which is actually defined directly in the gragh spectral domain by $\hat{g}(\lambda_l) = e^{-5\lambda_l}$ The signal plotted in (a) is then determined by taking an inverse grapgh Fourier transform (3.6) of $\hat{\mathbf{g}}$.

Interpretation

In classical Fourier analysis, the **eigenvalues** $\{(2\pi\xi)^2\}_{\xi\in\mathbb{R}}$ in (3.2) carry a specific notion of **frequency**: **for ξ close to zero (low frequencies), the associated complex exponential eigenfunctions are smooth**, slowly oscillating functions, whereas **for ξ far from zero (high frequencies), the associated complex exponential eigenfunctions oscillate much more rapidly**. In the graph setting, the graph Laplacian eigenvalues and eigenvectors provide a similar notion of frequency. For connected graphs, the Laplacian eigenvector ϕ_1 associated with the eigenvalue 0 is constant and equal to $\frac{1}{\sqrt{N}}$ at each vertex. **The graph Laplacian eigenvectors associated with low frequencies λ_l vary slowly across the graph**; i.e., if two vertices are connected by an edge with a large weight, the values of the eigenvector at those locations are likely to be similar. **The eigenvectors associated with larger eigenvalues oscillate more rapidly and are more likely to have dissimilar values on vertices connected by an edge with high weight**.

The graph Fourier transform (3.4) and its inverse (3.6) give us a way to equivalently represent a signal in two different domains: the spatial domain and the graph spectral domain. While we often start with a signal g in the spatial domain, it may also be useful to define a signal $\hat{\mathbf{g}}$ directly in the graph spectral domain. We refer to such signals as *kernels*. In Figures 6.13(a) and 6.13(c), one such signal, a *heat kernel*, is shown in both domains. Analogously to the classical analog case, the **graph Fourier coefficients** of a smooth signal such as the one shown in Figure 3.2 decay rapidly. Such signals are *compressible* as they can be closely approximated by just a few graph Fourier coefficients (see, *e.g.* [14, 63, 64] for ways to exploit this compressibility).

3.2.3 Discrete Calculus and Signal Smoothness

When we analyze signals, it is important to emphasize that properties such as smoothness are *with respect to the intrinsic structure of the data domain*, which in our context is the weighted grapgh (*i.e.* $\mathbf{W}$).

To add mathematical precision to the notion of smoothness with respect to the intrinsic structure of the underlying graph, we briefly present some of the discrete differential operators defined in [56] [2].

Definition 3.2.7 (edge derivative)**.** The *edge derivative* of a signal $\mathbf{f}$ with respect to edge $e = (i, j)$ at vertex i is defined as

$$\left.\frac{\partial \mathbf{f}}{\partial e}\right|_i := \sqrt{W_{ij}}[f(j) - f(i)]$$

Definition 3.2.8 (graph gradient)**.** The *graph gradient* of $\mathbf{f}$ at vertex i is the vector

$$\nabla_i \mathbf{f} := \left[\left\{ \left.\frac{\partial \mathbf{f}}{\partial e}\right|_i \right\}_{e \in \mathcal{E} \text{ s.t } e=(i,j) \text{ for some } j \in \mathcal{V}} \right]$$

Definition 3.2.9 (local variation)**.** We can then define *local variation* at vertex i.

$$\begin{aligned} ||\nabla_i \mathbf{f}||_2 : &= \left[\sum_{e \in \mathcal{E} \text{ s.t } e=(i,j) \text{ for some } j \in \mathcal{V}} \left(\left.\frac{\partial \mathbf{f}}{\partial e}\right|_i \right)^2 \right]^{\frac{1}{2}} \\ &= \left[\sum_{j \in \mathcal{N}_i} W_{ij}[f(j) - f(i)]^2 \right]^{\frac{1}{2}} \end{aligned}$$

provides a measure of **local smoothness** of $\mathbf{f}$ around vertex i, as it is small when the function $\mathbf{f}$ has similar values at i and all neighboring vertices of i.

Definition 3.2.10 (discrete p-Dirichlet form of $\mathbf{f}$)**.** For notions for **global smoothness**, the notion of *discrete p-Dirichlet form* of $\mathbf{f}$ is defined as

$$S_p(\mathbf{f}) := \frac{1}{p} \sum_{i \in \mathcal{V}} ||\nabla_i \mathbf{f}||_2^p = \frac{1}{p} \sum_{i \in \mathcal{V}} \left[\sum_{j \in \mathcal{N}_i} W_{ij}[f(j) - f(i)]^2 \right]^{\frac{p}{2}} \tag{3.7}$$

When $p = 1$, $S_1(\mathbf{f})$ is the *total variation* of the signal with respect to the graph. When $p = 2$, we have

$$\begin{aligned} S_2(\mathbf{f}) &= \frac{1}{2} \sum_{i \in \mathcal{V}} ||\nabla_i \mathbf{f}||_2^p = \frac{1}{p} \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{N}_i} W_{ij}[f(j) - f(i)]^2 \\ &= \sum_{i,j \in \mathcal{E}} W_{ij}[f(j) - f(i)]^2 \\ &= \mathbf{f}^\mathsf{T} \mathbf{\Delta} \mathbf{f} \end{aligned} \tag{3.8}$$

[2] Note that the names of many of the discrete calculus operators correspond to the analogous operators in the continuous setting. In some problems, the weighted graph arises from a discrete sampling of a smooth manifold. In that situation, the discrete differential operators may converge - possibly under additional assumptions - to their namesake continuous operators as the density of the sampling increases.

$S_2(\mathbf{f})$ is also known as the *graph Laplacian quadratic form* [57].

Returning to the graph Laplacian eigenvalues and eigenvecotrs, the Courant-Fischer Theorem [28] tells us they can also be defined iteratively via the Rayleigh quotient as

$$\lambda_1 = \min_{\substack{\mathbf{f}\in\mathbb{R}^N \\ ||\mathbf{f}||_2=1}} \{\mathbf{f}^\mathsf{T}\mathbf{\Delta}\mathbf{f}\} \tag{3.9}$$

$$\text{and } \lambda_l = \min_{\substack{\mathbf{f}\in\mathbb{R}^N \\ ||\mathbf{f}||_2=1 \\ \mathbf{f}\perp \text{Span}(\phi_1,\ldots,\phi_N)}} \{\mathbf{f}^\mathsf{T}\mathbf{\Delta}\mathbf{f}\},\ l=2,3,\ldots,N \tag{3.10}$$

where the eigenvector ϕ_l is the minimizer of the l^{th} problem. From (3.8) and (3.9), we see why ϕ_1 is constant for connected graphs. Equation (3.10) explains why the **graph Laplacian eigenvectors associated with lower eigenvalues are smoother**, and provides another interpretation for why the graph Laplacian spectrum carries a notion of frequency.

In summary, the connectivity of the underlying graph is encoded in the *graph Laplacian*, which is used to define both a *graph Fourier transform* (via the graph Laplacian eigenvectors) and different notions of *smoothness*.

3.2.4 Filtering

We start by extending the notion of frequency filtering to the graph setting, and then discuss localized filtering in the vertex (spatial) domain.

Frequency Filtering

In classical signal processing, frequency filtering is the process of representing an input signal as a linear combination of complex exponentials, and amplifying or attenuating the contributions of some of the component complex exponentials:

$$\hat{f}_{out}(\xi) = \hat{f}_{in}(\xi)\hat{h}(\xi) \tag{3.11}$$

where $\hat{h}(\xi)$ is the transfer function of the filter. Taking an inverse Fourier transform of (3.11), multiplication in the Fourier domain corresponds to convolution in the time domain:

$$f_{out}(t) = \int_{\mathbb{R}} \hat{f}_{in}(\xi)\hat{h}(\xi)e^{2\pi i\xi t}d\xi \tag{3.12}$$

$$= \int_{\mathbb{R}} f_{in}(\tau)h(t-\tau)d\tau := (f_{in} * h)(t) \tag{3.13}$$

Definition 3.2.11 (graph spectral filtering)**.** Once we fix a graph spectral representation, and thus our notion of a graph Fourier transform (in this section, we use the eigenvectors of $\mathbf{\Lambda}$, but $\widetilde{\mathbf{\Lambda}}$ can also be used), we can directly generalize (3.11) to define frequency filtering, or graph spectral filtering, as

$$\hat{f}_{out}(\lambda_l) = \hat{f}_{in}(\lambda_l)\hat{h}(\lambda_l) \tag{3.14}$$

or, equivalently, taking an inverse graph Fourier transform,

$$f_{out}(i) = \sum_{l=1}^{N} \hat{f}_{in}(\lambda_l)\hat{h}(\lambda_l)\phi_l(i) \tag{3.15}$$

By using the matrix notation, we can also write (3.14) and (3.15) as $\mathbf{f}_{out} = \hat{h}(\mathbf{\Delta})\mathbf{f}_{in}$, where

$$\hat{h}(\mathbf{\Delta}) := \mathbf{\Phi} \begin{bmatrix} \hat{h}(\lambda_1) & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \hat{h}(\lambda_N) \end{bmatrix} \mathbf{\Phi}^{\mathsf{T}} \tag{3.16}$$

Filtering in the Vertex Domain

To filter a signal in the vertex domain, we simply write the output $f_{out}(i)$ at vertex i as a linear combination of the components of the input signal at vertices within a K**-hop local neighborhood** of vertex i:

$$f_{out}(i) = b_{i,i}f_{in}(i) + \sum_{j \in \mathcal{N}(i,K)} b_{i,j}f_{in}(j) \tag{3.17}$$

for some constants $\{b_{i,j}\}_{i,j\in\mathcal{V}}$. Equation (3.17) just says that filtering in the vertex domain is a localized linear transform.

Relation of filtering in the graph spectral domain and the vertex domain

We now briefly relate filtering in the graph spectral domain (frequency filtering) to filtering in the vertex domain. When the frequency filter in (3.14) is an order K polynomial $\hat{h}(\lambda_l) = \sum_{k=0}^{K} a_k\lambda_l^k$ for some constants $\{a_k\}_{k=0,1,\ldots,K}$, we can also interpret the filtering equation (3.14) in the vertex domain. From (3.15), we have

$$\begin{aligned} f_{out}(i) &= \sum_{l=1}^{N} \hat{f}_{in}(\lambda_l)\hat{h}(\lambda_l)\phi_l(i) \\ &= \sum_{j=1}^{N} f_{in}(j) \sum_{k=0}^{K} a_k \sum_{l=1}^{N} \lambda_l^k \phi_l^*(j)\phi_l(i) \\ &= \sum_{j=1}^{N} f_{in}(j) \sum_{k=0}^{K} a_k(\mathbf{\Delta}^{\mathbf{k}})_{i,j} \end{aligned} \tag{3.18}$$

Yet, $(\mathbf{\Delta}^{\mathbf{k}})_{i,j} = 0$ when the shortest-path distance $d_{\mathcal{G}}(i,j)$ between vertices i and j ((i.e. the minimum number of edges comprising any path connecting i and j) is greater than k [23]. Therefore, we can write (3.18) exactly as in (3.17), with the constants defined as

$$b_{i,j} = \sum_{k=d_{\mathcal{G}}(i,j)}^{K} a_k(\mathbf{\Delta}^k)_{i,j}$$

So **when the frequency filter is an order K polynomial**, the frequency filtered signal at vertex i, $f_{out}(i)$, is a linear nation of the components of the input signal at vertices within a K**-hop local neighborhood** of vertex i. This property can be quite useful when relating the smoothness of a filtering kernel to the localization of filtered signals in the vertex domain. We will see how this property was used to construct spatial localized filter in next chapter.

3.2.5 Graph Convolution

We cannot directly generalize the definition 3.13 of a convolution product to the graph setting, because of the term $h(t\tau)$. However, one way to define a generalized convolution product for signals on graphs is to replace the complex exponentials in (3.12) with the graph Laplacian eigenvectors [55].

Definition 3.2.12 (Graph Convolution)**.** Graph convolution is defined with the graph Laplacian eigenvectors as:

$$(f * h)(i) := \sum_{l=1}^{N} \hat{f}(\lambda_l)\hat{h}(\lambda_l)\phi_l(i) \tag{3.19}$$

Or in a matrix notation:

$$\begin{aligned} \mathbf{f} * \mathbf{h} &= \mathbf{\Phi}\mathrm{diag}(\hat{h}(\lambda_1), \ldots, \hat{h}(\lambda_N))\mathbf{\Phi}^{\mathsf{T}}\mathbf{f} \\ &= \hat{h}(\mathbf{\Delta})\mathbf{f} \end{aligned} \tag{3.20}$$

The matrix notation of graph convolution is what we most used in the spectral graph CNNs. We can notice that filter coefficients **depend on Laplacian eigenvectors** $\phi_1, \ldots, \phi_N$**,** which property makes the filter learned in one graph difficult to transfer to another graph. We can also find that the **computation complexity** is $\mathcal{O}(n^2)$ rather than $\mathcal{O}(n)$ in classical Convolution in spatial domain. We will elaborate these in detail in the next chapter.

3.2.6 Translation

Convolution in spatial domain have the good property of **translation invariance**. We then inspect this property in graphs. The classical translation operator is defined through the change of variable $(T_v f)(t) := f(t - v)$, which, as discussed earlier, we cannot generalized directly to graph setting. However, we can also view the classical translation operator T_v as a convolution with a delta centred at v; i.e. $(T_v f)(t) = (f * \delta_v)(t)$ in the weak sense. Thus, one way to define a *generalized translation operator* $T_N : \mathbb{R}^N \to \mathbb{R}^N$ is via generalized convolution with a delta centered at vertex n [23, 55].

Definition 3.2.13 (generalized translation)**.** As discussed above, we can then define generalized translation operator:

$$(T_n g)(i) := \sqrt{N}(g * \delta_n)(i) = \sqrt{N}\sum_{l=1}^{N} \hat{g}(\lambda_l)\phi_l^*(n)\phi_l(i) \tag{3.21}$$

where

$$\delta_n(i) = \begin{cases} 1 & \text{if } i = n \\ 0 & \text{otherwise} \end{cases} \tag{3.22}$$

A few remarks about the generalized translation (3.21) are in order:

1. We do not usually view it as translating a signal g defined in the vertex domain, but rather as a kernelized operator acting on a kernel $\hat{g}(\cdot)$ defined directly in the graph spectral domain.

2. The normalizing constant $\sqrt{N}$ in (3.21) ensures that the translation operator preserves the mean of a signal.

3. The smoothness of the kernel $\hat{g}(\cdot)$ controls the localization of $T_n g$ around the centre vertex n; that is, the magnitude $(T_n g)(i)$ of the translated kernel at vertex i decays as the distance between i and n increases [23]. This property can be seen in Figure 3.3, where we translate a heat kernel around to different locations of the Minnesota graph.

4. Unlike the classical translation operator, the generalized translation operator (3.21) is not generally an isometric operator ($||T_n \mathbf{g}||_2 \neq ||\mathbf{g}||_2$), due to the possible localization of the graph Laplacian eigenvectors ($\phi_l(i) > \frac{1}{\sqrt{N}}$).

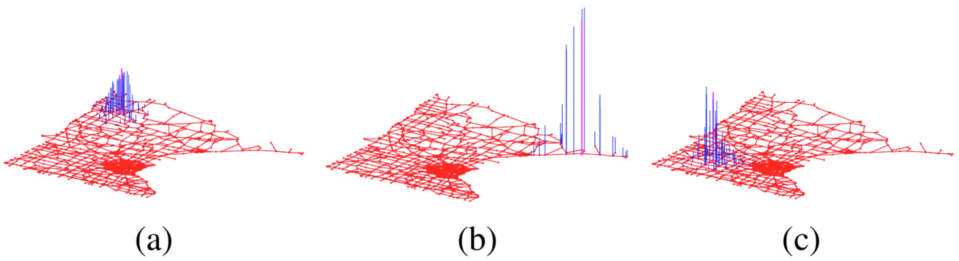


Figure 3.3: The translated signals (a) $T_1 00\mathbf{g}$, (b) $T_2 00\mathbf{g}$, and (c) $T_2 000\mathbf{g}$, where $\mathbf{g}$ is the heat kernel shown in Figures 6.13(a) and 6.13(c)

3.3 Graph Coarsening

We recall the concept of pooling in classical CNNs discussed in Section 2.1.2. Basically, the pooling layer serves three main purposes: 1) By having less spatial information we can gain computation performance; 2) Less spatial information also means less parameters, so less chance to overfitting; 3) we can get some invariance to global geometric deformations. Hence, we also want to find pooling methods operating on graphs that preserve properties lied in classical CNNs.

Typically, the process of transforming a given graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathbf{W}\}$ into a coarser graph $\mathcal{G}^{reduced} = \{\mathcal{V}^{reduced}, \mathcal{E}^{reduced}, \mathbf{W}\}$ with fewer vertices and edges, while also preserving

intrinsic geometric structures, is often referred to as *graph coarsening* or *coarse-graining* [36]. From this process, we can find grouped vertices and manage pooling operations on that.

Graph clustering is an important problem with many applications, and a number of different algorithms and methods have emerged over the years. Spectral methods have been used effectively for solving a number of graph clustering objectives, including *ratio cut* [8] and *normalized cut* [54]. Such an approach has been useful in many areas, such as circuit layout [8] and image segmentation [54]. However, such spectral methods that compute k eigenvectors require $\mathcal{O}(nk)$ storage, where n is the number of data points and k is the number of clusters, is inefficient for large-scale tasks. We introduce a highly efficient algorithm, called *Graclus* [16], which is used for graph clustering. We should stress that the goal of pooling is to **reduce the graph size**, and **there is no need to perform formal clustering stage**. It is enough If the graph size can be reduced in some reasonable way, this is why we only perform *graph coarsening*. In typical clustering problems, people do graph coarsening in the first stage to obtain a smaller graph and then classify a smaller set of vertices. To sum up, we would focus on the first stage – **graph coarsening** of the *Graclus algorithm*

Multi-level Graclus Algorithm

The framework of Graclus algorithm is similar to that of Karypis and Kumar [30], a popular multi-level graph clustering algorithm for optimizing the *Kernighan-Lin objective*. Figure 3.4 provides a graphical overview of the multilevel framework. It includes three phases: coarsening, base clustering, and refinement. We recommend readers who are interested in the whole algorithm procedures to [16]. Below we describe the *coarsening phase* of the framework.

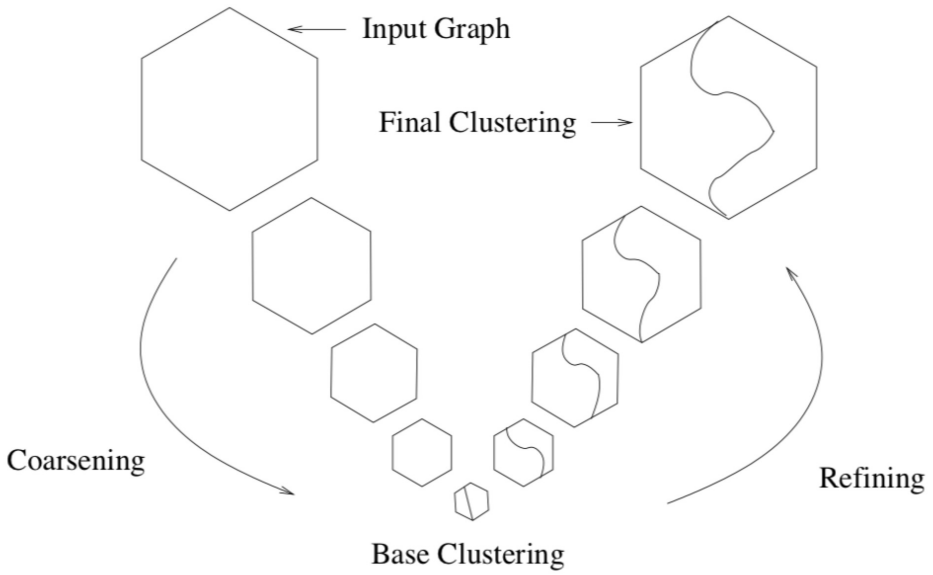


Figure 3.4: Overview of the multi-level algorithm (for $k = 2$).

Coarsening Phase Starting with the initial graph $\mathcal{G}_0$, the coarsening phase repeatedly transforms the graph into smaller and smaller graphs $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m$ such that $|\mathcal{V}_0| > |\mathcal{V}_1| > \ldots > |\mathcal{V}_m|$. To coarsen a graph from $\mathcal{G}_i$ to $\mathcal{G}_{i+1}$, sets of nodes in $\mathcal{G}_i$ are combined to form supernodes in $\mathcal{G}_{i+1}$. When combining a set of nodes into a single supernode, the edge weights out of the supernode are taken to be the sum of the edge weights out of the original nodes.

The coarsening works as follows: given a graph, start with all nodes unmarked. Visit each vertex in a random order. For each vertex x, if x is not marked, merge x with the unmarked vertex y that maximizes (3.23) among all edges between x and unmarked vertices. Then mark x and y. If all neighbors of x have been marked, mark x and do not merge it with any vertex. Once all vertices are marked, the coarsening for this level is compete.

$$\max \; W_{x,y} \left(\frac{1}{d_x} + \frac{1}{d_y} \right) \tag{3.23}$$

$W_{x,y}$ corresponds to the edge weight between vertices x and y and d_x, d_y are the vertex degree of x and y, respectively.

This is a very fast coarsening scheme which divides the number of nodes by approximately two from one level to the next coarser level.

Chapter 4

Spectral and Spatial Graph Convolutions and Evaluation

In this chapter, we discuss *geometric deep learning* methods from both *spectral* and *spatial* perspective, where the goal is to apply deep learning on graph- or mesh- structured data instead of regular grids (*i.e.* images). We begin with defining *general spectral Graph CNNs (sGCNNs)* and *general spatial Graph CNNs*, and argue that most of the well-known methods in this field (*e.g.* Bruna et al. [7], Henaff et al. [26], Defferrard et al. [15], Kipf and Welling [34], Masci et al. [42], Boscaini et al. [5], Monti et al. [44], Fey et al. [18]) can be constructed from these two frameworks. Then, we have an in-depth discussion of these networks, pointing out their design intuition, insights, and key properties.

4.1 Spectral Convolution Operations

4.1.1 General Spectral Graph CNNs

Notations and Definitions

We are interested in analyzing signals defined on an undirected, connected, weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$, which consists of a finite set of vertices $\mathcal{V}$ with $|\mathcal{V}| = n$, a set of edges $\mathcal{E}$, and a weighted adjacency matrix $\mathbf{W} \in \mathbb{R}^{n \times n}$. If there is an edge $e = (i, j)$ connecting vertices i and j, the entry W_{ij} (or a_{ij}) represents the weight of the edge; otherwise, $W_{ij} = 0$.

A *signal* or *vertex function* $f : \mathcal{V} \to \mathbb{R}$ defined on the vertices of the graph may be represented as a vector $\mathbf{f} \in \mathbb{R}^n$, where the i^{th} component of the vector $\mathbf{f}$ represents the function value at the i^{th} vertex in $\mathcal{V}$.

We recall the definitions in Section 3.2 here. The *non-normalzied graph Laplacian* is $\mathbf{\Delta} = \mathbf{D} - \mathbf{W}$, where the *degree matrix* $\mathbf{D} = \operatorname{diag}(\sum_j W_{ij})$, and *normalized definition* is $\widetilde{\mathbf{\Delta}} = \mathbf{D}^{-1/2}\mathbf{\Delta}\mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}$. Because the graph Laplacian $\mathbf{\Delta}$ is a real symmetric matrix, it has a complete set of orthogonal eigenvectors, which we denote by $\mathbf{\Phi} = (\phi_1, \ldots, \phi_n)$[1], known as the *graph Fourier basis*. These eigenvectors have associated

[1]Note that there is not necessarily a unique set of graph Laplacian eigenvectors, but we assume throughout that a set of eigenvectors is chosen and fixed.

real, non-negative eigenvalues $\lambda_1, \ldots, \lambda_n$, identified as the frequencies of the graph, satisfying $\mathbf{\Delta}\phi_i = \lambda_i \phi_i$, for $i = 1, 2, \ldots, n$. The *graph Fourier transform* of a signal $\mathbf{f} \in \mathbb{R}^n$ is then defined as $\hat{\mathbf{f}} = \mathbf{\Phi}^\mathsf{T}\mathbf{f}$, and its inverse as $\mathbf{f} = \mathbf{\Phi}\,\hat{\mathbf{f}}$.

Then the *Graph Convolution* is defined as:

$$\mathbf{f} * \mathbf{h} = \mathbf{\Phi} \begin{bmatrix} \hat{h}(\lambda_1) & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \hat{h}(\lambda_N) \end{bmatrix} \mathbf{\Phi}^\mathsf{T}\mathbf{f} \tag{4.1}$$

$$= \hat{h}(\mathbf{\Delta})\mathbf{f} \tag{4.2}$$

where $\mathbf{h}$ is the convolutional filter on graphs.

Architecture

Similar to classical CNNs (Sec. 2.1), a spectral graph CNN architecture is commonly formed by a stack of distinct layers (CONV, POOL, FC). The distinction between various approaches lies in, (1) the design of the spectral convolutional filters on graphs, (2) a graph coarsening procedure that groups together similar vertices and a graph pooling operation that trades spatial resolution for higher filter resolution (see Figure 4.1).

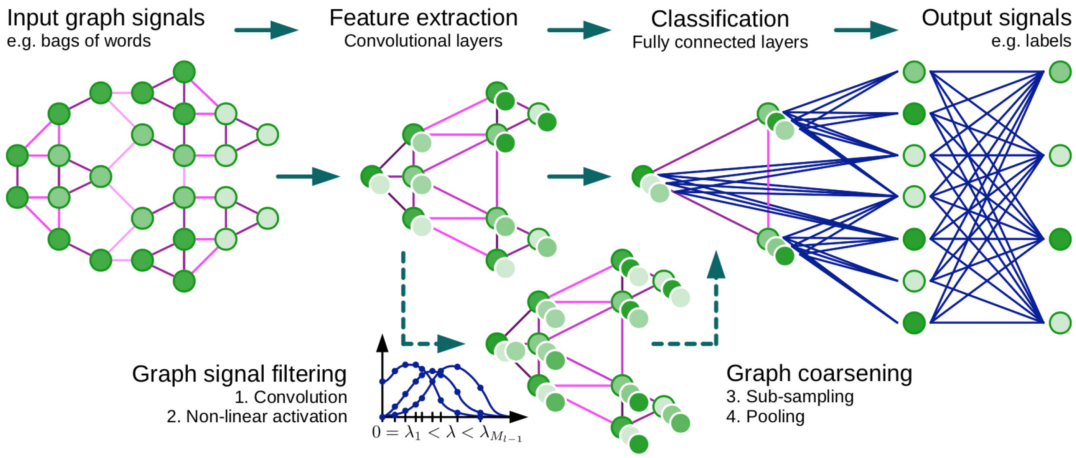


Figure 4.1: Architecture of a CNN on graphs. (Figure reproduced from [15])

Convolutional Layer Similar to the convolutional layer (2.1) of a classical Euclidean CNNs, we can define convolution for each layer as

$$\mathbf{g}_{l'} = \xi \left(\sum_{l=1}^{p} \mathbf{\Phi} \begin{bmatrix} \hat{h}_{l,l'}(\lambda_1) & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \hat{h}_{l,l'}(\lambda_n) \end{bmatrix} \mathbf{\Phi}^\mathsf{T}\mathbf{f}_l \right) \tag{4.3}$$

$$= \xi \left(\sum_{l}^{p} \mathbf{\Phi} \widehat{\mathbf{H}}_{l,l'} \mathbf{\Phi}^{\mathsf{T}} \mathbf{f}_l \right) \tag{4.4}$$

the vectors $\{\mathbf{f}_l : \mathbf{f}_l \in \mathbb{R}^n\}_{l=1,\dots,p}$ and $\{\mathbf{g}_{l'} : \mathbf{g}_{l'} \in \mathbb{R}^n\}_{l'=1,\dots,q}$ represent the p- and q-dimensional input and output signals defined on the vertices of the graph, respectively. $\widehat{\mathbf{H}}_{l,l'}$ represents the spectral trainable filter, and ξ is a non-linear function applied on the vertex-wise value.

We should stress that the progress of spectral-based networks lay on **the design of the spectral filter**, which influence the computational complexity and the number of parameters in the convolutional layer and then the network performance. In the following section, we focus on this part of each method, and show the advantages and maintaining drawbacks.

Pooling Layer The pooling layer in the sGCNN is typically along with *graph coarsening* process (details in Sec. 3.3). We recall the definition of *general pooling* defined in (2.2),

$$g'_l(x) = P(\{f_l(x') : x' \in \mathcal{N}(x)\}), \quad l = 1, \dots, q$$

where $\mathcal{N}(x) \subset \mathcal{V}$ is a neighborhood around x and P is a permutation-invariant function such as a L_p-norm. We therefore resort to *graph coarsening* techniques to construct such neighborhood at different resolutions. This process transforms a given graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, \mathbf{W}\}$ into a coarser graph $\mathcal{G}^{reduced} = \{\mathcal{V}^{reduced}, \mathcal{E}^{reduced}, \mathbf{W}\}$ with fewer vertices and edges. Basically, for each vertex i in the coarsest graph $\mathcal{G}^{reduced}$, we treat the corresponding vertex set $\mathcal{A}_i \in \mathcal{V}$ as the neighbors, and then perform *max pooling*.

Fast Pooling After coarsening, however, a direct application of the pooling operation would need a table to store all matched vertices, which would result in a memory inefficient, slow, and hardly parallelizable implementation. Defferrard et al. [15] proposed a fast-pooling approach by arranging the vertex indexing such that adjacent nodes are hierarchically merged at the next coarser level. This makes it as efficient as a $1D$-Euclidean grid pooling. We briefly explain the method here:

1. **Create the balanced binary tree.** After coarsening in each level, each vertex has either two children if it was matched at the original graph, otherwise one. For those single vertices, fake vertices are added to pair with the singletons such that each node has two children. This makes a *balanced binary tree*. Input signals are initialized with a neutral value at the fake nodes, *e.g.* 0.

2. **Rearrange the vertex.** Arbitrarily ordering the nodes at the coarsest level, then propagating this ordering to the original levels, *i.e.* node k has nodes $2k$ and $2k+1$ as children, produces a regular ordering in the original level. Adjacent vertices then are hierarchically merged at coarser levels.

Figure 4.2 shows an example of the whole process. This regular arrangement makes the operation very efficient and satisfies parallel architectures such as GPUs as memory accesses are local, *i.e.* matched nodes do not have to be fetched.

In the following sections, we will not focus on coarsening and pooling methods here and instead will **emphasize high-level differences that exist across different networks**.

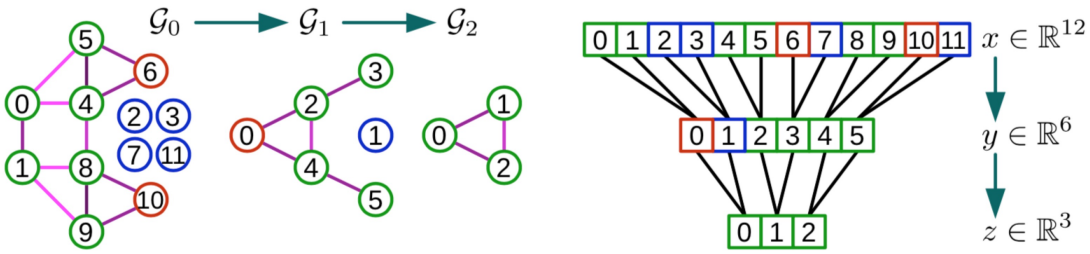


Figure 4.2: Example of Graph Coarsening and Pooling. We carry out a max pooling of size 4 on a signal $\mathbf{x} \in \mathbb{R}^8$ on $\mathcal{G}_0$, the original graph given as input. Note that it originally possesses $n_0 = |\mathcal{V}_0| = 8$ vertices, arbitrarily ordered. For a pooling of size 4, two coarsenings of size 2 are needed: let Graclus gives G1 of size $n_1 = |\mathcal{V}_1| = 5$, then $\mathcal{G}_2$ of size $n_2 = |\mathcal{V}_2| = 3$,the coarsest graph. Sizes are thus set to $n_2 = 3, n_1 = 6, n_0 = 12$ and fake vertices (in blue) are added to $\mathcal{V}_1$ (1 vetex) and $\mathcal{V}_0$ (4 vertex) to pair with the singeltons (in orange), such that each vertex has exactly two children. Vertices in $\mathcal{V}_2$ are then arbitrarily ordered and vertices in $\mathcal{V}_1$ and $\mathcal{V}_0$ are ordered consequently. At that point the arrangement of vertices in $\mathcal{V}_0$ permits a regular $1D$ pooling on $\mathbf{x} \in \mathbb{R}^{12}$ such that $z = [\max(\mathbf{x}(0), \mathbf{x}(1)), \max(\mathbf{x}(4), \mathbf{x}(5), \mathbf{x}(6)), \max(\mathbf{x}(8), \mathbf{x}(9), \mathbf{x}(10))] \in \mathbb{R}^3$, where the signal components $\mathbf{x}(2)$, $\mathbf{x}(3)$, $\mathbf{x}(7)$, $\mathbf{x}(11)$ are set to a neutral value.

For convenience, We consider *Graclus* [16], multilevel clustering algorithm (Sec. 3.3), combined with *fast pooling* approach [15] as a **General Graph Pooling Layer** in this thesis.

Notes on optimization and implementation details

Most of experiments are set on the task of classification or signal prediction on vertices of graphs. Spectral network architectures are commonly based on a *classical convolutional network*, namely by interleaving *graph convolution*, *ReLU* and *graph pooling layers* (no this layer if the task is for vertex classification), and ending with one or more *fully connected layers*. Models are trained with *cross-entropy loss*, using *SGD+Momentum* and *Adam* [32] optimization methods.

4.1.2 Vanilla Spectral Graph CNNs

Conv Layer

Bruna et al. [7] (2013) pioneered the work to generalize a convolutional layer by operating on the spectrum of the weights, given by the eigenvectors of its graph Laplacian:

$$\mathbf{g}_{l'} = \xi \left(\sum_{l=1}^{p} \mathbf{\Phi}_d \mathrm{diag}(\alpha_{1,l,l'}, \ldots, \alpha_{d,l,l'}) \mathbf{\Phi}_d^{\mathsf{T}} \mathbf{f}_l \right), \; l = 1, \ldots, p; \; l' = 1, \ldots, q \tag{4.5}$$

where $\mathrm{diag}(\alpha_{1,l,l'}, \ldots, \alpha_{d,l,l'})$ is a $d \times d$ diagonal matrix of spectral multipliers presenting as a learnable filter in the frequency domain, and $\mathbf{\Phi}_d$ represents the first d Laplacian eigenvectors sorted by eigenvalues from lowest to highest ($\lambda_1 \leq \ldots \leq \lambda_d \leq \ldots \leq \lambda_n$). Only first d eigenvectors are used, because in practice, **the first d eigenvectors of the Laplacian are useful which carry the smooth geometry of the graph**. We provide the explanation

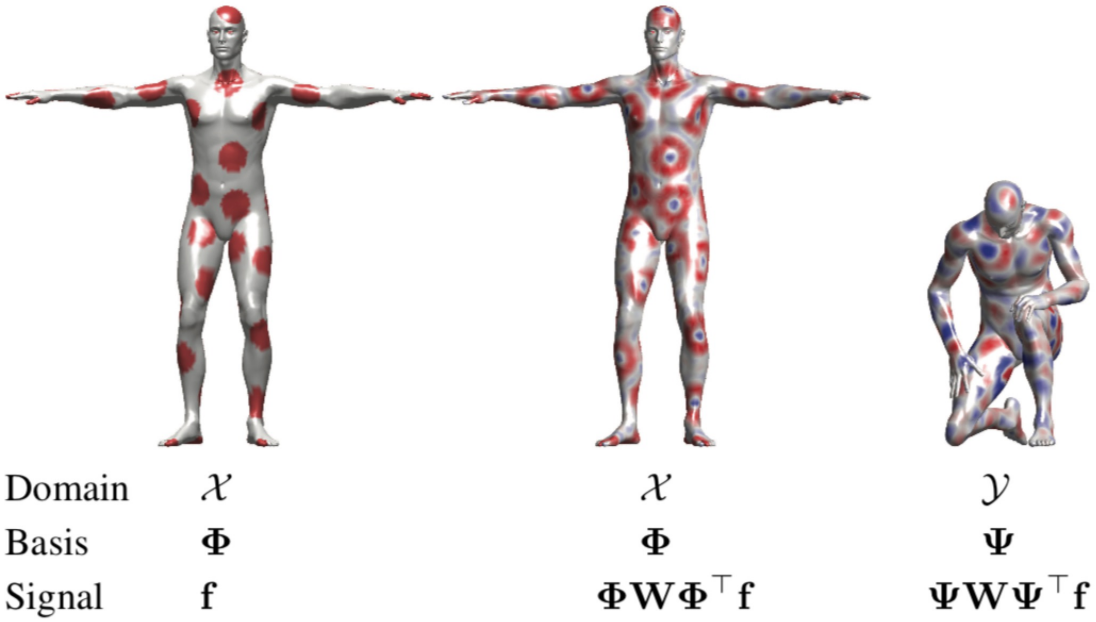


Figure 4.3: A toy example illustrating the difficulty of generalizing spectral filtering across non-Euclidean domains. Left: a function defined on a manifold (function values are represented by color); middle: result of the application of an edge-detection filter in the frequency domain; right: the same filter applied on the same function but on a different (nearly-isometric) domain produces a completely different result. The reason for this behavior is that the Fourier basis is domain-dependent, and the filter coefficients learnt on one domain cannot be applied to another one in a straightforward manner. The figure is from Bronstein et al. [6].

in Section 3.2.2, that is, *the graph Laplacian eigenvectors associated with low frequencies* λ_l *vary slowly across the graph, i.e. smooth or localized in the spacial domain.* This is a very important property when we try to use *harmonic analysis* on graphs. Generally, the cutoff frequency d depends upon the intrinsic regularity of the graph and also the sample size.

Bruna et al. [7] argued that if the graph has an underlying group invariance this spectral construction can discover it. They showed that, when applied to natural images, the construction in (4.5) using the covariance as the similarity kernel recovers a standard convolutional network, without any prior knowledge, which can also implies **translation invariant** in the classical Euclidean CNNs. However, in many cases the graph does not have a group structure, or the group structure does not compute with the Laplacian, and so we cannot think of each filter as passing a template across $\mathcal{V}$ and recording the correlation of the template with that location. Bruna et al. [7] proved this from the **subsampled MNIST** experiment that in the spectral construction the measurements are not enforced to become spatially localized and therefore cannot recognize localized oriented strokes of MNIST digits effectively. By adding the smoothness constraint on the spectrum of the filters improves classification results, since the filters are then enforced to have better spatial localization.

Pooling Layer

The pooling in this framework is obtained via dropping the last part of the spectrum of the Laplacian, leading to max-pooling. It follows that strided convolutions can be generalized using the spectral construction by keeping only the low-frequency components of the spectrum. This property allows us to interpret (via interpolation) the local filters at deeper layers in the spatial construction to be low frequency.

In this work, they only used a naive multiscale clustering on the space side construction that is not guaranteed to respect the original graph's Laplacian and no explicit spatial clustering in the spectral construction.

Complexity Analysis

Firstly, we consider about the complexity of the learnable parameters in each layer. Assuming only d eigenvectors of the Laplacian are kept, equation (4.5) shows that each layer requires $p \cdot q \cdot d = \mathcal{O}(|\mathcal{V}|)$ paramters to train, whereas we should stress that, in classical ConvNets, this is $\mathcal{O}(1)$. Complex models may lead to overfitting to some extent. This is one of the **drawbacks** of this method.

Nevertheless, the computational complexity in each layer can not be neglected. The computation of the forward and inverse graph Fourier transformation incurs expensive $\mathcal{O}(|\mathcal{V}|^2)$ multiplication by the matrices $\mathbf{\Phi}_k$ and $\mathbf{\Phi}_k^{\mathsf{T}}$, as there is no FFT-like algorithms on general graphs. By comparison, the classical ConvNets requires $\mathcal{O}(n)$ computation in each conv layer. This therefore is another **drawback** of this approach.

Drawbacks

In summary, there are four distinct drawbacks of such methods (N.B. the analysis of pooling procedure is not included, and the same as the following other compared methods):

1. $\mathcal{O}(|\mathcal{V}|)$ trainable parameters in each conv layer.
2. $\mathcal{O}(|\mathcal{V}|^2)$ computational complexity in each conv layer.
3. Spectral filters are not guaranteed for spacial localization.
4. Filters are basis-dependent, *i.e.* not generalized across graphs.

4.1.3 SplineNets

In order to make the convolutional kernels restricted to have small spatial support, which enables the model to learn a number of parameters independent of the input size and also reduce the risk of overfitting, Henaff et al. [26] (2015) argued that smooth spectral filter coefficients result in spatially-localized filters (an argument similar to *vanishing moments*). The theory support derives from the *Parseval Identity*,

$$\int_{-\infty}^{+\infty} |x|^{2k} f(x)^2 dx = \int_{-\infty}^{+\infty} |\frac{\partial^k \hat{f}(\lambda)}{\partial \lambda^k}|^2 d\lambda \tag{4.6}$$

This suggests that, in order to learn a layer in which features will be not only shared across locations but also well localized in the original domain, one can learn spectral multipliers which are smooth. Bruna et al. [7] suggested to use this as principle in a general graph, by considering a smoothing kernel $\mathcal{K} \in \mathbb{R}^{N \times N_0}$, such as *cubic splines*.

However, the notion of *smoothness* requires some geometry in the spectral domain. We explain the smoothness with respect to the intrinsic structure of the weighted graph (*i.e.* $\mathbf{W}, \mathcal{G}$) in Sec. 3.2.3. The notions of **global smoothness** is defined through *discrete p-Dirichlet form* of $\mathbf{f}$, and in particular, $p = 2$,

$$S_2(\mathbf{f}) = \mathbf{f}^\mathsf{T} \mathbf{\Delta} \mathbf{f}$$

Conv Layer

Henaff et al. [26] used the spectral filter as the form of

$$\hat{h}(\lambda_i) = \sum_{j=1}^{r} \alpha_j \beta_j(\lambda_i),\ i = 1, 2, \ldots, k \tag{4.7}$$

where $\beta_1(\lambda_i), \ldots, \beta_r(\lambda_i)$ are some fixed interpolation kernels such as cubic splines, and $\alpha_1, \ldots, \alpha_r$ are learanble parameters. In matrix notation, the conv layer can then be expressed as

$$\mathbf{g}_{l'} = \xi \left(\sum_{l}^{p} \mathbf{\Phi}_k \mathrm{diag}(\mathbf{B}\alpha) \mathbf{\Phi}_k^\mathsf{T} \mathbf{f}_l \right)\ l = 1, \ldots, p;\ l' = 1, \ldots, q \tag{4.8}$$

where $\mathbf{B} = (\beta_j(\lambda_i))$ is a $k \times r$ matrix and $\alpha = (\alpha_1, \ldots, \alpha_r)$ is a vector of coefficients.

Pooling Layer

The similar multi-resolution spectral clustering was used here with that of Bruna et al. [7] Vanilla Spectral Graph CNNs.

Complexity Analysis

As the interpolation coefficients of spline are fixed to r, the total number of learnable parameters in each lay is independent of the input size $\mathcal{V}$ of the graph $\mathcal{G}$, thus recovering the same learning complexity as CNNs on Euclidean grids. Hence, We can expect the improvements of the performance.

The computation complexity remain $\mathcal{O}(|\mathcal{V}|^2)$ because of the computation of forward and backward Fourier transformation, *i.e.* $\mathbf{\Phi}_k, \mathbf{\Phi}_k^\mathsf{T}$.

Drawbacks

Remaining drawbacks can then be concluded as,

1. $\mathcal{O}(|\mathcal{V}|^2)$ computational complexity in each conv layer.
2. Filters are basis-dependent.

4.1.4 ChebNets

We recall an important **relation of filtering in the graph spectral domain and the vertex domain** illustrated in Sec. 3.2.4. When the frequency filter in (3.14) is an order of K polynomial $\hat{h}(\lambda_l) = \sum_{k=0}^{K} a_k \lambda_l^k$ for some constants $\{a_k\}_{k=0,1,\ldots,K}$, we can interpret the filtering equation (3.14) in the vertex domain. We have

$$f_{out}(i) = \sum_{j=1}^{N} f_{in}(j) \sum_{k=0}^{K} a_k (\mathbf{\Delta^k})_{i,j}$$

Moreover, given $(\mathbf{\Delta^k})_{i,j} = 0$ when the shortest-path distance $d_{\mathcal{G}}(i,j)$ between vertices i and j (*i.e.* the minimum number of edges comprising any path connecting i and j) is greater than k [23], we can observe an useful property that **if the spectral filter is an order of K polynomial, it is exactly K-hop localized in the spatial domain**.

Conv Layer

Defferrard et al. [15] (2016) exploited this property and designed localized filters of the form of polynomial parametrization.

$$\hat{h}(\lambda_i) = \sum_{j=0}^{r-1} \alpha_j \lambda_i^j, \; i = 1, \ldots, n \tag{4.9}$$

In matrix notation,

$$\hat{h}(\Lambda) = \begin{bmatrix} \sum_{j=0}^{r-1} \alpha_j \lambda_1^j & \cdots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \cdots & \sum_{j=0}^{r-1} \alpha_j \lambda_n^j \end{bmatrix} \tag{4.10}$$

Then the convolution form is

$$\mathbf{g}_{l'} = \xi \left(\sum_{l}^{p} \mathbf{\Phi} \hat{h}(\Lambda) \mathbf{\Phi}^{\mathsf{T}} \mathbf{f}_l \right) \tag{4.11}$$

$$= \xi \left(\sum_{l}^{p} \hat{h}(\mathbf{\Delta}) \mathbf{f}_l \right), \; l = 1, \ldots, p; \; l' = 1, \ldots, q \tag{4.12}$$

However the computational complexity of equation (4.12) is still high with $\mathcal{O}(|\mathcal{V}|^2)$ operations because of the multiplication with the Fourier basis Φ. A solution to this problem is to parametrize $\hat{h}(\mathbf{\Delta})$ as a polynomial function that can be computed recursively from $\mathbf{\Delta}$, as r multiplications by a sparse graph $\mathcal{G}$ costs $\mathcal{O}(r|\mathcal{E}|) \ll \mathcal{O}(|\mathcal{V}|^2)$. One such polynomial, traditionally used in *Graph Signal Processing* to approximate kernels (like wavelets), is the **Chebyshev polynomial** [23].

Definition 4.1.1 (Chebyshev polynomial)**.** The **Chebyshev polynomials** are defined by the recurrence relation

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \end{aligned}$$

These polynomials form an orthogonal basis for $L^2([-1,1], dy/\sqrt{1-y^2})$, the Hilbert space of square integrable functions with respect to the measure $dy/\sqrt{1-y^2}$.

Chebyshev Spectral Filter Defferrard et al. [15] designed the spectral filter using *Chebyshev polynomial*. The filter can thus be parametrized as the truncated expansion

$$\hat{h}(\Lambda) = \sum_{j=0}^{r-1} \alpha_j T_j(\hat{\Lambda}) \tag{4.13}$$

where the parameter $\{\alpha_j\}_{j=0,\dots,r-1}$ are the coefficients of Chebyshev polynomials. $\hat{\Lambda} = 2\Lambda/\lambda_{max} - I$, a diagonal matrix of scaled eigenvalues that lie in $[-1,1]$. We can see the Figure 4.4 that it is stable under perturbation of the coefficients. The convolution operation can then be written as

$$\mathbf{g}_{l'} = \xi \left(\sum_{l}^{p} \mathbf{\Phi} \sum_{j=0}^{r-1} \alpha_j T_j(\hat{\Lambda}) \mathbf{\Phi}^{\mathsf{T}} \mathbf{f}_l \right) \tag{4.14}$$

$$= \xi \left(\sum_{l}^{p} \sum_{j=0}^{r-1} \alpha_j T_j(\hat{\Delta}) \mathbf{f}_l \right) \; l = 1, \dots, p;\; l' = 1, \dots, q \tag{4.15}$$

where $\hat{\mathbf{\Delta}} = 2\mathbf{\Delta}/\lambda_{max} - I$. Denoting $\mathbf{F}_{j,l} = T_j(\hat{\mathbf{\Delta}})\mathbf{f}_l \in \mathbb{R}^n$, we can then use the recurrence relation to compute $\mathbf{F}_{j,l} = 2\hat{\mathbf{\Delta}}\mathbf{F}_{j-1,l} - \mathbf{F}_{j-2,l}$ with $\mathbf{F}_{0,l} = \mathbf{f}_l$ and $\mathbf{F}_{1,l} = \hat{\mathbf{\Delta}}\mathbf{f}_l$. The equation (4.15) can then be written as

$$\mathbf{g}_{l'} = \xi \left(\sum_{l}^{p} \sum_{j=0}^{r-1} \alpha_j \mathbf{F}_{j,l} \right) \; l = 1, \dots, p;\; l' = 1, \dots, q \tag{4.16}$$

Pooling Layer

Pooling layer consists of two steps: (i) the coarsening phase of *multilevel Graclus' algorithm*; (ii) Defferrard et al. [15] fast pooling method.

Complexity Analysis

Cayley spectral filters now reduce computational complexity to $\mathcal{O}(r|\mathcal{E}|) = \mathcal{O}(n)$ for sparse graphs, as there is no need to compute the forward and backward Fourier transform, and also eigendecomposition of the Laplacian Δ.

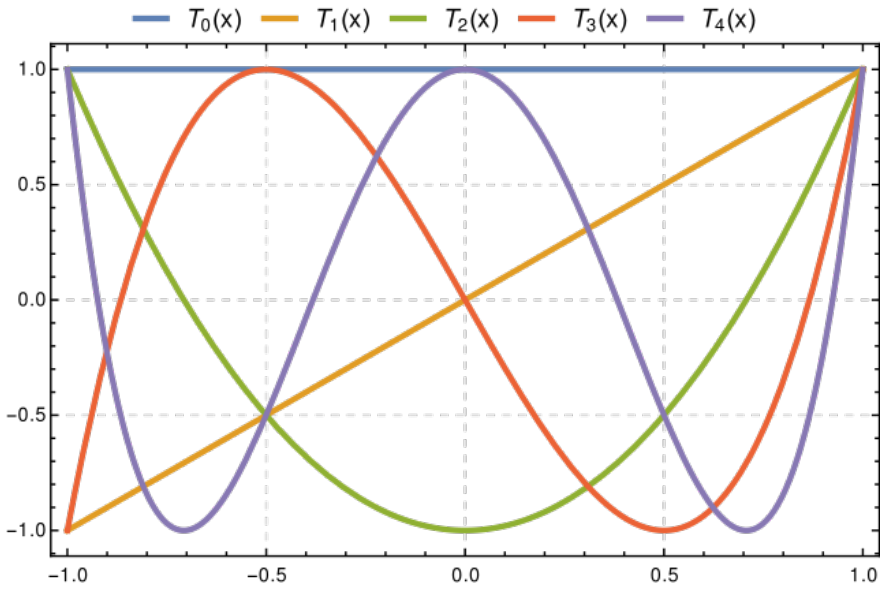


Figure 4.4: Plot of the first five Chebyshev T polynomials

Drawbacks

This spectral construction still have the drawback of basis dependent.

4.1.5 GraphConvNets

Kipf and Welling [34] (2016) simplified the Chebyshev polynomial of order $r-1$ to the linear form, *i.e.* $r=2$, and assumed $\lambda_{max}=2$. Under these approximations, equation (4.15) simplifies to:

$$\mathbf{g}_{l'} = \xi\left(\sum_{l}^{p} \alpha_0 \mathbf{f}_l + \alpha_1(\boldsymbol{\Delta} - I)\mathbf{f}_l\right) \tag{4.17}$$

$$= \xi\left(\sum_{l}^{p} \alpha_0 \mathbf{f}_l - \alpha_1 \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}\mathbf{f}_l\right) \quad l=1,\ldots,p;\ l'=1,\ldots,q \tag{4.18}$$

with two free parameters α_0 and α_1. The Laplacian Δ here means the normalized graph Laplacian.

Intuition The intuition can be explained that such a model can alleviate the problem of **overfitting** on local neighborhood structures for graphs with very wide node degree distributions, such as social networks, citation networks and many other real-world graph datasets. Additionally, for a fixed computational budget, this layer-wise linear formulation allows to build deeper models, a practice that is known to improve modeling capacity on a number of domains (He et al. [25]).

In practice, Kipf and Welling [34] further constrained $\alpha=\alpha_0=-\alpha_1$ to address **overfitting** and to minimize the number of operations per layer. This simplifies equation (4.18)

to the following expression:

$$\mathbf{g}_{l'} = \xi\left(\sum_{l}^{p} \alpha(I - \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2})\mathbf{f}_l\right) \; l = 1, \ldots, p;\; l' = 1, \ldots, q \tag{4.19}$$

As the eigenvalues of $(I - \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2})$ are now in the range $[0, 2]$, repeated application of such operator can lead to numerical instabilities and gradient vanishing when used in a deep neural network model. This can be alleviated by further *renormalization*,

$$\mathbf{g}_{l'} = \xi\left(\sum_{l}^{p} \alpha\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{W}}\tilde{\mathbf{D}}^{-1/2}\mathbf{f}_l\right) \; l = 1, \ldots, p;\; l' = 1, \ldots, q \tag{4.20}$$

where $\tilde{\mathbf{W}} = \mathbf{W} + I$ and $\tilde{\mathbf{D}} = \sum_j \tilde{W}_{ij}$

4.2 Spatial Graph Convolution Operations

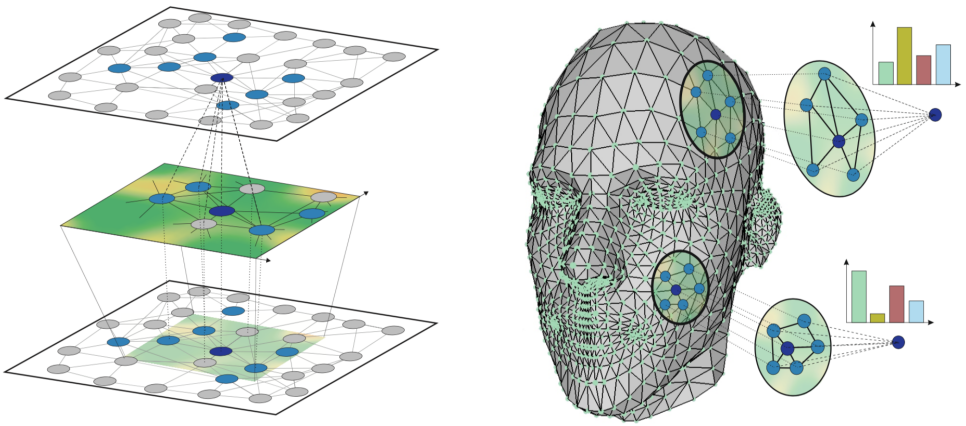
The issues of spectral graph CNNs (1. the model trained on one shape is hard to be applied to another shape; 2. spectral filters cannot capture local positional relations) can be tackled by using methods that extract representations for local Euclidean neighborhoods from discrete manifolds. Because of this property, the spatial convolution techniques are considered as better choices when dealing with 3D shape analysis such as shape correspondence on FAUST dataset. It is worth noting that in the computer graphics community, 3D shapes are normally modelled as manifolds (further discretized as meshes).

To our best knowledge, Masci et al. [42] proposed the first intrinsic version of convolutional neural networks on manifolds applying filters to local patches represented in geodesic polar coordinates. Boscaini et al. [5] improve this approach by introducing anisotropic heat kernels. Monti et al. [43] then proposed a more general framework (MoNet), allowing to design convolutional deep architectures on non-Euclidean domains such graphs and manifolds. Finally, Fey et al. [18] introduced SplineCNN, which leverages properties of B-spline bases to efficiently filter geometric input of arbitrary dimensionality. Currently, SplineCNN achieved the state-of-the-art result on the problem shape correspondence on FAUST dataset. We would introduce these four convolution methods in the following sections.

4.2.1 General Spatial Graph Convolution

Notations and Definitions

In the domain of computer graphics, 3D shapes are typically modeled as Riemannian manifolds and are discretized as meshes. Since mesh can be further generalized as graphs, we define the input data as an undirected, connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{U})$, which consists of a finite set of vertices $\mathcal{V}$ with $|\mathcal{V}| = n$, a set of edges $\mathcal{E}$, and d-dimensional pseudo-coordinates $\mathbf{u}(i, j) \in \mathbb{R}^d$. If there is an edge $e = (i, j)$ connecting vertices i and j, the pseudo-coordinate $\mathbf{u}(i, j)$ represents the corresponding *edge attribute*, for example, local Carteisian coordinate for 3D meshes or vertex degree for general graphs; otherwise, $\mathbf{u}(i, j) = 0$.



(a) Spatial convolution on graphs (b) Spatial Convolution on meshes

Figure 4.5: Examples for spatial convolution in geometric deep learning for (a) image graph representations and (b) meshes.

A *signal* or *vertex function* $f : \mathcal{V} \to \mathbb{R}$ defined on the vertices of the graph may be represented as a vector $\mathbf{f} \in \mathbb{R}^n$, where the i^{th} component of the vector $\mathbf{f}$ represents the function value at the i^{th} vertex in $\mathcal{V}$.

Convolution

We follow the form of definition that we define convolution in Euclidean domain (see Equation (2.1)). For each convolutional layer of the form $\mathbf{g} = \mathcal{C}_\Gamma(\mathbf{f})$, acting on a p-dimensional input $\mathbf{f}(i) = (f_1(i), \ldots, f_p(i))$ by applying a bank of filters $\Gamma = (\gamma_{l,l'}(\mathbf{u}(i,j)))$, $l = 1, \ldots, p, l' = 1, \ldots, q, \mathbf{u}(i,j) \in \mathbb{R}^d$, where $f_k(i)$ represents the k^{th} feature channel at the i^{th} vertex in $\mathcal{V}$. The output feature at i^{th} vertex is then defined as:

$$g_{l'}(i) = \sum_{l=1}^{p} \sum_{j \in \mathcal{N}(i)} f_l(i) \cdot \gamma_{l,l'}(\mathbf{u}(i,j)) \; l' = 1, \ldots, q \tag{4.21}$$

For compact support filters, the space complexity of parameters in per filter is $\mathcal{O}(1)$ (independent of input image size n) and the computational complexity is $\mathcal{O}(|\mathcal{E}|)$. Generally, the filter functions Γ defines the way of how to model the pseudo-coordinates $\mathbf{u}(i,j)$ (normally referred as Kernel) and how to parametrize those kernels. We stress that different frameworks for non-Euclidean spatial CNNs essentially amount to different choice of these kernels. Figure 4.5 explains how pseudo-coordinates are aggregated during convolution.

4.2.2 GeodesicCNN

Masci et al. [42] firstly proposed the idea by employing a local system of geodesic polar coordinates constructed at point x to extract patches on the manifold. It is natural to come up with this idea since in shape analysis, 3D shapes can be modelled as 2-dimensional surface, which allows us to create a polar system of coordinates around x where the radial

coordinate is given by some intrinsic distance $\rho(i,j) = d(i,j)$, and the angular coordinate $\theta(i,j)$ is obtained by ray shooting from a point at equispaced angles.

Now, the filter function Γ defined in (4.21) has the Gaussian kernel 4.22 and the pseudo-coordinates $\mathbf{u}(i,j)$ is defined as the local polar geodesic coordinates $p(i,j), \theta(i,j)$ on a $2D$ surface:

$$\mathcal{K}_{\alpha,\beta}(\rho_{i,j}, \theta_{i,j}) = e^{-(\rho(j)-\rho_\alpha)^2/2\sigma_\alpha^2} \, e^{-(\theta(j)-\theta_\beta)^2/2\sigma_\beta^2} \tag{4.22}$$

Where $\alpha = 1, \ldots, J$ and $\beta = 1, \ldots, J'$ denote the indices of the radial and angular bins, respectively. Then, the filter function is defined as:

$$\gamma_{l,l'}(\rho_{i,j}, \theta_{i,j}) = \sum_{\alpha,\beta} \mathcal{W}_{\alpha,\beta} \, \mathcal{K}_{\alpha,\beta}(\rho_{i,j}, \theta_{i,j}) \tag{4.23}$$

where $\mathcal{W}_{\alpha,\beta}$ represents weights operating on each kernel. The resulting $\alpha \cdot \beta$ weights are bins of width $\sigma_\rho \cdot \sigma_\theta$ in the polar coordinates.

Geodesic Convolution we can now define the geodesic convolution layer by substituting 4.23 into the general form of spatial convolution 4.21, yielding:

$$g_{l'}(i) = \sum_{l=1}^{p} \sum_{j \in \mathcal{N}(i)} f_l(i) \cdot \sum_{\alpha,\beta} \mathcal{W}_{\alpha,\beta} \, e^{-(\rho(j)-\rho_\alpha)^2/2\sigma_\alpha^2} \, e^{-(\theta(j)-\theta_\beta)^2/2\sigma_\beta^2} \tag{4.24}$$

where $l' = 1, \ldots, q$; $\alpha = 1, \ldots, J$; $\beta = 1, \ldots, J'$.

Discussion

Since GCNN operate in the spatial domain and thus do not suffer from the inherent inability of spectral methods to generalize across different domains. On the other hand, it works in an intrinsic way which is then considered to be more robust to capture local invariant information in 3D shape analysis.

4.2.3 AnisotropicCNN

Boscaini et al. [5] proposed Anisotropic Convolution Neural Netwokrs(ACNN), which beats the previous framework GCNN on changeling correspondence benchmarks. They argue that there are some drawbacks for GCNN. First, the method may fail if the mesh is very irregular. Second, the radius of the geodesic patches must be sufficiently small compared to the injectivity radius of the shape, otherwise the resulting patch is not guaranteed to be a topological disk. In practice, this limits the size of the patches one can sfaely use, or requires an adpative radius selection mechanism.

The construction of ACNN inherits all the advantages of the aforementioned intrinsic CNN approaches, without holding their drawbacks.

Anisotropic Convolution

The idea of the Anisotropic CNN is the construction of a patch operator using anosotropic heat kernels. The *anisotropic diffusion* equation on the manifold is defined as

$$f_t(x,t) = -\mathrm{div}_{\mathcal{X}}(\mathbf{A}(x)\nabla_{\mathcal{X}} f(x,t)) \tag{4.25}$$

where $\nabla_{\mathcal{X}}$ and $\mathrm{div}_{\mathcal{X}}$ denote the *intrinsic gradient* and *divergence*, respectively, $f(x,t)$ is the temperature at point x and time t, and the *conductivity tensor* $\mathbf{A}(x)$ (operating on the gradient vectors in the tangent space $\mathbf{T}_x\mathcal{X}$) allows to model heat flow that is position- and direction-dependent. In particular, they used the 2×2 tensor

$$\mathbf{A}_{\alpha\theta}(x) = \mathbf{R}_\theta(x) \begin{pmatrix} \alpha & \\ & 1 \end{pmatrix} \mathbf{R}_\theta^\mathsf{T}(x) \tag{4.26}$$

where matrix $\mathbf{R}_\theta$ performs rotation of θ w.r.t to some reference (*e.g.* the maximum curvature) direction and $\alpha > 0$ is a parameter controlling the degree of anisotropy ($\alpha = 1$ corresponds to the classical isotropic case). Using as initial condition $f(x,0)$ a point source of heat at x, the solution to the heat equation (4.25) is given by the *anisotropic heat kernel* $h_{\alpha\theta t}(x,)$, representing the amount of heat that is transferred from point x to point y at time t.

Now, we can define the kernel of ACNN as the form of our definition for general spatial convolution (4.21).

$$\mathcal{K}_\beta(\mathbf{u}(i,j)) = \exp(-\frac{1}{2}\mathbf{u}^\mathsf{T}\mathbf{R}_{\theta_\beta} \begin{pmatrix} \alpha & \\ & 1 \end{pmatrix} \mathbf{R}_{\theta_\beta}^\mathsf{T}\mathbf{u}) \tag{4.27}$$

where $\beta = 1, \ldots, d$; d denotes to the dimensionality of the manifolds.

So the anisotropic convolutional layer is defined as:

$$g_{l'}(i) = \sum_{l=1}^{p} \sum_{j \in \mathcal{N}(i)} f_l(i) \cdot \sum_\beta \mathcal{W}_\beta \exp(-\frac{1}{2}\mathbf{u}(i,j)^\mathsf{T}\mathbf{R}_{\theta_\beta} \begin{pmatrix} \alpha & \\ & 1 \end{pmatrix} \mathbf{R}_{\theta_\beta}^\mathsf{T}\mathbf{u}(i,j)) \tag{4.28}$$

note that the pseudo-coordinates $\mathbf{u}(i,j)$ here denotes to *local polar coordinates* $\rho(i,j)$ and $\theta(i,j)$.

4.2.4 MoNet

Monti et al. [43] generalise the previous spatial domain framework for deep learning on non-Euclidean domains by introducing a local system of d-dimensional pseudo-coordinates $\mathbf{u}(i,j)$ around vertex i. They defined parametric kernels $\mathcal{K}_\beta(\mathbf{u}(i,j))$ instead of the previous fixed kernel constructions,

$$\mathcal{K}_\beta(\mathbf{u}(i,j)) = \exp(-\frac{1}{2}(\mathbf{u} - \mu_\beta)^\mathsf{T}\mathbf{\Sigma}_\beta^{-1}(\mathbf{u} - \mu_\beta)) \quad \beta = 1, \ldots, d \tag{4.29}$$

where $\mathbf{\Sigma}_\beta$ and μ_β are learnable $d \times d$ and $d \times 1$ covariance matrix and mean vector of a Gaussian kernel, respectively. Monti et al. [43] further restrict the covariances to have diagonal form, resulting in $2d$ parameters per kernel.

We can then conclude the convolutional layer of MoNet in our defined general spatial convolution framework as below:

$$g_{l'}(i) = \sum_{l=1}^{p} \sum_{j \in \mathcal{N}(i)} f_l(i) \cdot \sum_{\beta}^{d} \mathcal{W}_\beta \exp(-\frac{1}{2}(\mathbf{u} - \mu_\beta)^\intercal \mathbf{\Sigma}_\beta^{-1}(\mathbf{u} - \mu_\beta)) \tag{4.30}$$

where d denotes to the dimensionality of the manifolds or graphs. Clearly, the spacial complexity is $\mathcal{O}(d^2)$ and the computational complexity is $\mathcal{O}(|\mathcal{E}|)$. MoNet can also be applied on general graphs using the pseudo-coordinates $\mathbf{u}$ as the pseudo-coordinates $\mathbf{u}$ some local graph features such as vertex degree. Figure 4.6 shows how patch operator kernel functions $\mathcal{K}(\mathbf{u}(i,j))$ of GCNN, ACNN and MoNet are used in different generalizations of convolution on the manifold.

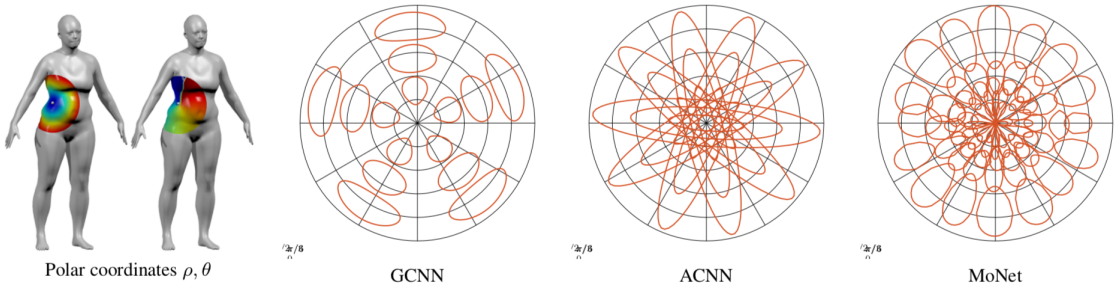


Figure 4.6: Left: intrinsic local polar coordinates , on manifold around a point marked in white. Right: patch operator kernel functions $\mathcal{K}(\mathbf{u}(i,j))$ used in different generalizations of convolution on the manifold (hand-crafted in GCNN and ACNN and learned in MoNet. Figure is from [43].

4.2.5 SplineCNN

Fey et al. [18] present *Spline-based Convolutional Neural Networks (SplineCNNs)* by leveraging properties of B-spline bases to efficiently filter geometric input of arbitrary dimensionality, which makes it currently the state-of-the-art approach in several applications. Similar to MoNet, SplineCNN takes use of the pseudo-coordinates $\mathrm{u}(i,j)$ as input. During locally aggregating feature values in a local patch, pseudo-coordinates determine how the features are aggregated and the content of $f(i)$ define what is aggregated.

Following our previous defnition for the general spatial convolution framework, the B-spline based kernel of SplineCNN can then be written as:

$$\mathcal{K}_{\mathbf{p} \in \mathcal{P}}(\mathbf{u}(i,j)) = \prod_{i=1}^{d} N_{i,p_i}^{m}(u_i) \tag{4.31}$$

where N_{i,p_i}^{m} denotes open B-spline bases of degree m, based on uniform, *i.e.* equidistant, knot vectors Piegl and Tiller [49]. Then the filter function Γ is defined as:

$$\gamma_{l,l'}(\mathbf{u}(i,j)) = \sum_{\mathbf{p}\in\mathcal{P}} \mathcal{W}_{\mathbf{p},l,l'} \cdot \mathcal{K}_{\mathbf{p}\in\mathcal{P}}(\mathbf{u}) \tag{4.32}$$

where $\mathcal{W}_{\mathbf{p},l,l'}$ denotes to control points and thus the learnable parameters in the deep learning paradigm. Given the filter function $\Gamma = (\gamma_{l,l'})$ and input node features $\mathbf{f}$, the output feature at i^{th} vertex is then defined as:

$$g_{l'}(i) = \sum_{l=1}^{p} \sum_{j\in\mathcal{N}(i)} f_l(i) \cdot \sum_{\mathbf{p}\in\mathcal{P}} \mathcal{W}_{\mathbf{p},l,l'} \cdot \prod_{i=1}^{d} N_{i,p_i}^{m}(u_i)\ l' = 1,\ldots,q \tag{4.33}$$

From the code Fey et al. [18] provided, the convolution should also be normalized by node degree and consider the bias. We modify the equation (4.33) as below:

$$g_{l'}(i) = \frac{1}{|\mathcal{N}(i)|} \sum_{l=1}^{p} \sum_{j\in\mathcal{N}(i)} f_l(i) \cdot \sum_{\mathbf{p}\in\mathcal{P}} \mathcal{W}_{\mathbf{p},l,l'} \cdot \prod_{i=1}^{d} N_{i,p_i}^{m}(u_i) + b_{l,l'}\ l' = 1,\ldots,q \tag{4.34}$$

Due to the local support property of B-splines, $\mathcal{K}_{\mathbf{p}} \neq 0$ only holds true for $(m+1)^d$ different vectors $\mathbf{p} \in \mathcal{P}$. We denote the vectors $\mathbf{p} \in \mathcal{P}$ with $\mathcal{K}_{\mathbf{p}} \neq 0$ to $\mathcal{P}(\mathbf{u}(i,j))$. Then, the formula (4.34) can be further simplified to,

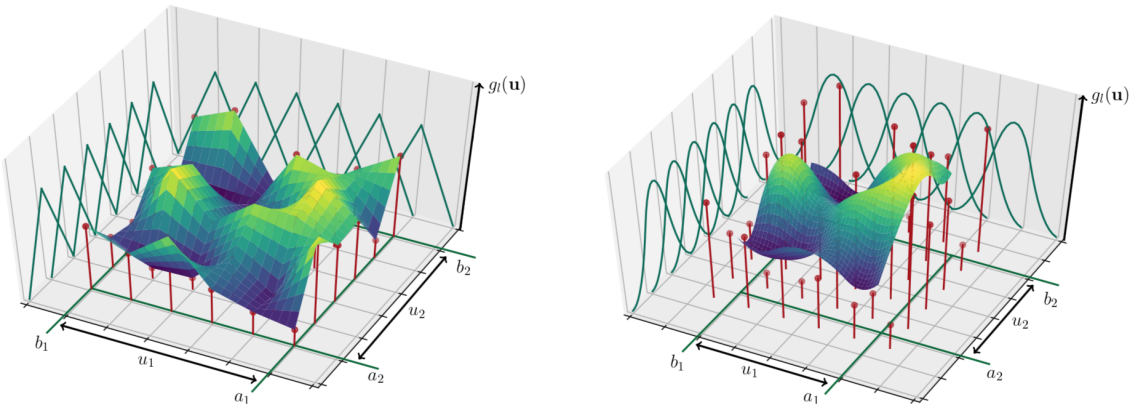
$$g_{l'}(i) = \frac{1}{|\mathcal{N}(i)|} \sum_{l=1}^{p} \Big(\sum_{j\in\mathcal{N}(i)} f_l(i) \sum_{\mathbf{p}\in\mathcal{P}(\mathbf{u}(i,j))} \mathcal{W}_{\mathbf{p},l,l'} \cdot \prod_{i=1}^{d} N_{i,p_i}^{m}(u_i) + b_{l,l'} \Big)\ l' = 1,\ldots,q \tag{4.35}$$

clearly, the trainable parameters amount to $l \cdot l' \cdot (m+1)^d = \mathcal{O}(1)$. Fey et al. [18] provides the visualization (see Figure 4.7) of the kernel construction method for differing B-spline basis degree m. They suggest that degree $m = 1$ produce better results no matter input data is general graphs or manifolds. We follow this setting in the thesis.

4.3 Evaluation

In order to evaluate the performance of various convolution operators we introduced in the previous sections, we implement them to tackle two representative tasks: (a) *Graph Vertex Classification* on *Cora Citation Network* [53], where the publication's content is represented as a sparse binary bag-of-word vector defined on top of nodes, and nodes are further structured as a graph; (b) *Shape Correspondence* on FAUST dataset [4], where 3D mesh-structured data provided, representing high-resolution human scans. From the evaluation on these two problems, we know the capacity, performance of different convolution operators, which is the evidence for why only the specified operators are chosen in the thesis when we develop models for the problems of *Protein Function Prediction* and *3D Facial Expression Analysis*.

Theoretically, Table 4.1 concludes and compares the key properties of spectral graph convolutions introduced in Sections (4.1.2, 4.1.3, 4.1.4, 4.1.5) and convolutions in euclidean domains. Since generally protein interaction networks are quite large ($5 - 30k$



(a) Linear B-spline basis functions

(b) Quadratic B-spline basis functions

Figure 4.7: Examples of B-spline basis degrees (a) $m = 1$ and (b) $m = 2$ for kernel dimensionality $d = 2$. The heights of the red dots are the trainable parameters for a single input feature map. They are multiplied by the elements of the B-spline tensor product basis before influencing the kernel value. (The figure is from [18])

vertices and $100k - 3M$ edges), we cannot afford $\mathcal{O}(N)$ space complexity and $\mathcal{O}(N^2)$ computational complexity. Therefore, we mainly focus on GCN and ChebNets. It's worth to explore the GPU memory usage, computational efficiency, the capacity of encoding or aggregating graph information of these two operators. Additionally, we'll also pay attention on the performance under different r-hops setup. This allows a to acquire knowledge about these two operators and design suitable hyperparameter settings when we move on to the problem of protein function prediction.

	Locality	Space Complexity	Computational Complexity
CNN (Euclidean)	Yes	$\mathcal{O}(1)$	$\mathcal{O}(N)$
Vanilla Spectral Graph CNN	No	$\mathcal{O}(N)$	$\mathcal{O}(N^2)$
SplineNets	Yes	$\mathcal{O}(1)$	$\mathcal{O}(N^2)$
ChebNets	**Yes** (r-hops)	$\mathcal{O}(\mathbf{1})$	$\mathcal{O}(\boldsymbol{N})$
GCN	**Yes** (1-hop)	$\mathcal{O}(\mathbf{1})$	$\mathcal{O}(\boldsymbol{N})$

Table 4.1: Comparison between Euclidean CNN and spectral graph convolution operators. We assume the graph is sparse where $\mathcal{O}(|\mathcal{E}|) = \mathcal{O}(|\mathcal{V}|)$, so there's a constant maximum number of edges per vertex. Here we denote $N = |\mathcal{V}|$.

As for spatial mesh convolution, Table 4.2 compares spatial mesh convolutional operators introduced in Sections (4.2.2, 4.2.3, 4.2.4, 4.2.5) in terms of the receptive field considered, input node descriptor and loss function during training and inference. Clearly, SplineCNN allows to perform end-to-end mesh training without hand-crafted feature descriptors as input. These modifications reduce the computation time and memory con-

sumption that are required to preprocess the data by a wide margin. Additionally, the loss function (4.36) used by GCNN, ACNN and MoNet is well-designed for shape correspondence this problem, which would punish geodesically far-way predictions stronger than predictions near the ground-truth nodes, while loss function used in SplineCNN is a less geometrically meaningful criterion, which adds the possibility for us to generalize the operator to other 3D shape analysis problems, such as 3D facial expression recognition. Therefore, we mainly focus on SplineCNN.

$$\ell_{\text{reg}}(\Theta) = -\sum_{(x,y^*(x))\in\mathcal{T}} \log f_\Theta(x, y^*(x)) \tag{4.36}$$

where $f_\Theta(x, y^*(x))$ denotes the probability of x mapped to y, and $y_*(x)$ denotes the ground-truth correspondence of x on the reference shape. $\mathcal{T} = (x, y^*(x))$ represents the collection x on the reference shape and $y^*(x)$. The advantages of this loss function in terms of punishing some geodesically wrong nodes can be interpreted by the truth that it inherits a part of *binary cross entropy* loss.

	Receptive Field	Node Descriptor (dimension)	Loss Function
GCNN	2-dim Geodesic Patch	OSD ($150d$) [40]	Multinomial Regression
ACNN	2-dim Geodesic Patch	SHOT ($544d$) [51]	Multinomial Regression
MoNet	2-dim Geodesic Patch	SHOT ($544d$) [51]	Multinomial Regression
SplineCNN	3-dim Manifolds	No ($1d$)	Cross Entropy

Table 4.2: Comparison between different spatial mesh convolutional operators. For SplineCNN, it doesn't need a hand-crafted descriptor for each node, while OSD or SHOT descriptors are considered to have more information about intrinsic shape context [35].

To sum up, the convolution operators evaluated in the following sections are shown in the Table. We implemented these operators to perform two mentioned tasks, *i.e.* graph node classification and shape correspondence.

	Input Data			
	Node Feature	Topological Structure	Pseudo-coordinates	Hyperparameters
ChebNets	Yes	Yes	No	polynomial degree r
GCN	Yes	Yes	No	No
SplineCNN	Yes	Yes	Yes	dim d, basis degree m

Table 4.3: Indication of the required input data and hyperparameters for the selected graph convolutional operators.

4.3.1 Semi-Supervised Graph Node Classification

We evaluate the performance of selected convolutional operators on the popolar Cora dataset, a citation network [53]. The Cora network consists of 2708 nodes denoted as scientific publications, which are classified into one of seven classes. The citation network consists of 5429 linkes. We treat the citation links as undirected edges. Each node in the network is described by a 1433 dimensional sparse binary word vector, where $0/1$-valued word indicating the absense/presence of the corresponding word from the dictionary.

Architecture and Experimental Setup

For all the experiments, we use the network architecture introduced in [34, 15, 43] of two convolutional layers with 7 outputs on the second layer. The ReLU activation function is applied at the output of the first layer. Training was done with 500 nodes and we use the remaining 2,208 nodes for testing, to simulate labeled and unlabeled information. We train all models for 200 epochs using Adam [32] with a learning rate of 0.01 and $L2$ regularization 0.005. We implemented models based on Pytorch. Models are trained and evaluated with one GPU NVIDIA GTX 1080. As for SplineCNN, we compute the pseudo-coordinates $\mathbf{u}(i,j)$ with the globally normalized degree of target nodes:

$$u(i,j) = \frac{deg(j)}{\max_{v \in \mathcal{V}} deg(v)}$$

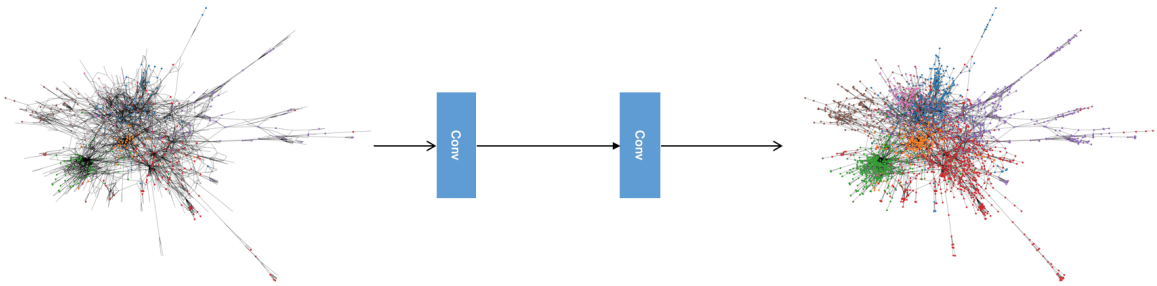


Figure 4.8: Architecture for the problem graph vertex classification based on Cora citation network. Left: given the input network with 500 labelled nodes colored with the groundtruth class; Right: predictions obtained applying graph convolution over the dataset, where marker fill color represents the predicted class; marker outline color represents the groundtruth class. The prediction network is produced with B-SplineConv.

Results

Hyperparameter Selection Firstly, we explore the optimial hyperparameters for ChebNets, GCN and SplineCNN with respect to this problem. We set the outputs of hidden layers as 16. Additionally, we did another comparison experiment using only fully connected layer (FC), which operation is unable to aggregate information with the help of the graph topological structure. Table 4.4 shows (i) both ChebyConv and B-SplineConv perform better with lower degree; (ii) All the three operators which take use of the topological structure information consistently achieved higher prediction accuracy than only fully connected operation on each layer. We will point out this later when we compare the method we developed on protein function prediction and deepNF which the underlying architecture is denoising autoencoder based only on fully connected layer. Figure 4.9 shows the consistant results with Table 4.4, but we also noticed that larger degree allows model to convergence faster. We can now make the following conclusions:

- From the results provided in Table 4.4 and Figure 4.9, the prediction accuracy over the task of semi-supervised graph vertex classification would see obvious improvement by simply replacing basic layer from fully connected layer to graph convolution (*e.g.* GCN, ChebyConv, B-SplineConv).
- Models with less complexity and less parameters perform consistently better.

Method	Filter	Hyperparameters	Accuracy
GCN (Eq. 4.20)	$\alpha\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{W}}\tilde{\mathbf{D}}^{-1/2}\mathbf{f}_l$	No	$\mathbf{85.51}\%$
ChebyConv (Eq. 4.16)	$\sum_{j=0}^{r-1}\alpha_j T_j(\hat{\Lambda})\mathbf{\Phi}^{\mathsf{T}}\mathbf{f}_l$	$r=2$	$\mathbf{85.37}\%$
		$r=3$	84.74%
B-SplineConv (Eq. 4.35)	$\sum_{j\in\mathcal{N}(i)} f_l(i)\sum_{\mathbf{p}\in\mathcal{P}}\mathcal{W}_{\mathbf{p},l,l'}\cdot\prod_{i=1}^{d}N_{i,p_i}^{m}(u_i)$	$m=1,k=2$	$\mathbf{85.78}\%$
		$m=2,k=3$	84.69%
		$m=3,k=4$	84.38%
FC	$\mathbf{F}_l\cdot\mathbf{W}_{l,l'}$	No	66.98%

Table 4.4: Summary of results in terms of classification accuracy. Displayed accuracies are averaged over 10 experiments, where for each experiment the network was trained for 200 epochs.

Neural Network Parameter Selection We further discuss about the influence of hidden layer outputs. This is a fundamental step to know about the potential capacity of each convolution operators under the situation when the model complexity becomes higher. Now, We initialize models with hidden layer outputs ranging from $16, 32, 128, 256$. We take

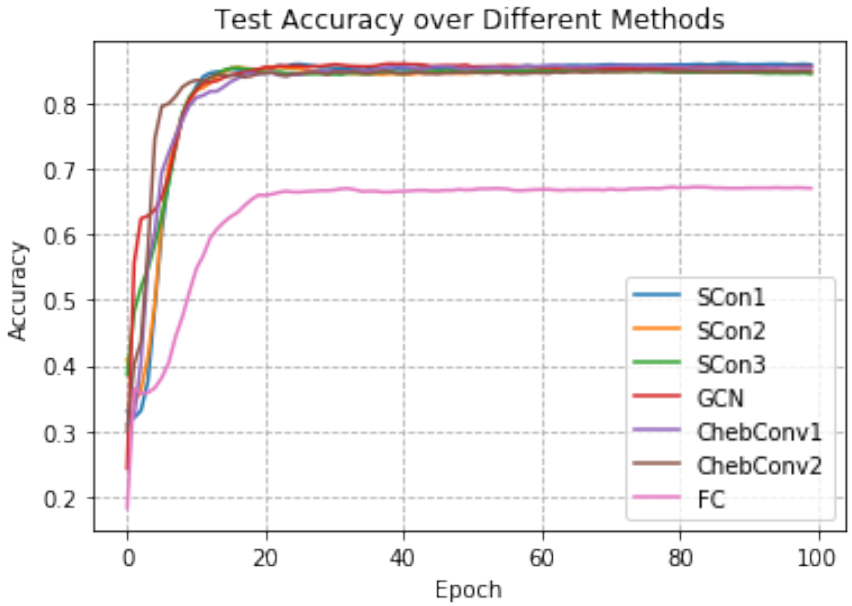


Figure 4.9: Test Accuracy over B-splineConv, GCN and ChebyConv on Cora Citation Network.

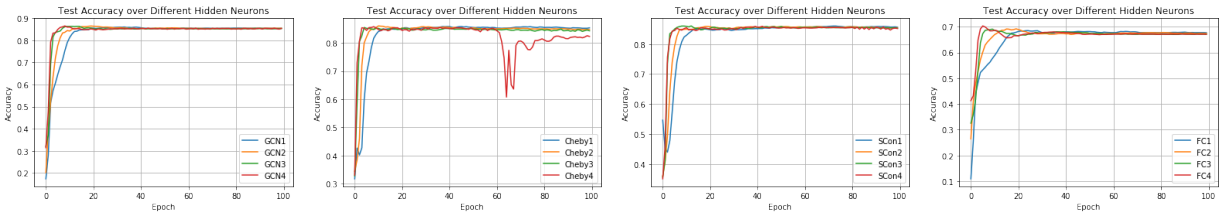
the optimial hyperparameter setting for ChebConv and B-SplineConv from the previous conclusion. Other experimental setup remain the same with before.

Table 4.5 shows all the methods with less hidden layer outputs performs better. Figure 4.10 proves that higher model complexity (more neurons) means faster convergence rate.

Method	Accuracy (percnetage)			
	16 HLO	32 HLO	128 HLO	256 HLO
GCN (Eq. 4.20)	**85.46**	85.51	85.37	85.42
ChebyConv (Eq. 4.16)	**85.33**	84.74	84.28	82.29
B-SplineConv (Eq. 4.35)	**85.64**	85.46	85.28	85.19
FC	**67.66**	67.39	67.07	67.03

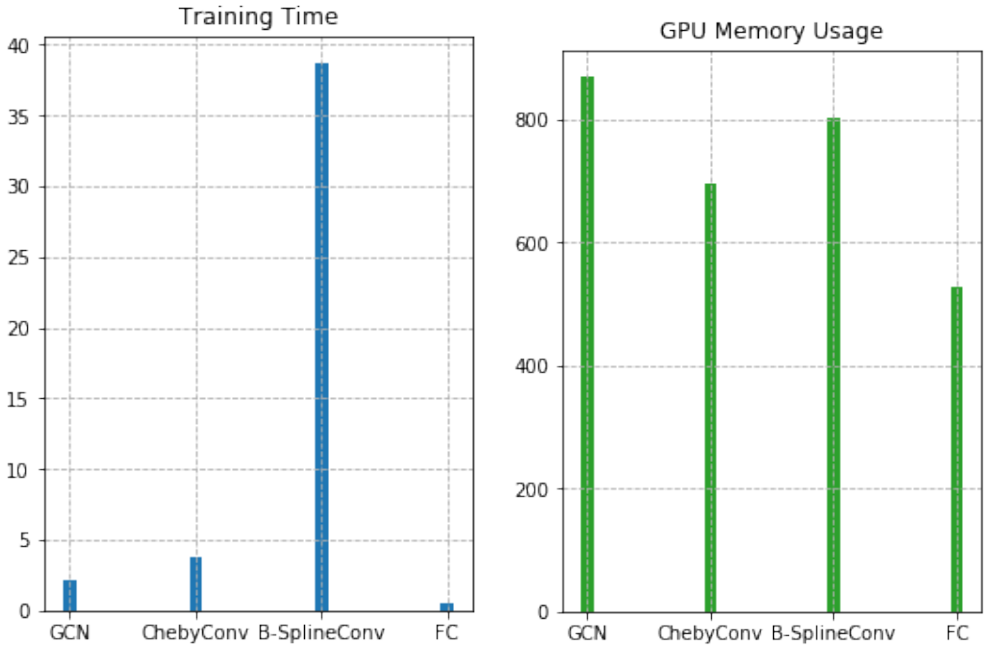
Table 4.5: Summary of results in terms of classification accuracy over different hidden layer outputs. Displayed accuracies are averaged over 10 experiments.

Training Time and GPU usage We test the training efficiency of all the methods and GPU usage under 256 hidden layer outputs. Figure 4.11 shows model training over B-SplineConv requires the highest time. Another distinct point is that although GCN is the simplification version of ChebyConv, it costs larger GPU memory during training process (note that ChebyConv compared in the bar chart use also the degree of 1 polynomial, which means 1-hop locality, so the comparison is fair).



(a) GCN (b) ChebyConv (c) B-SplineConv (d) FC

Figure 4.10: List of comparison of GCN, ChebyConv, B-SplineConv and FC over different hidden layer outputs.



(a) GCN (b) ChebyConv

Figure 4.11: GPU memory usage comparison between GCN, ChebyConv, B-SplineConv and FC. Training time is evaluated over *second* unit and GPU memory usage is over *MB*.

4.3.2 3D Shape Correspondence

Finding the correspondence in a collection of shapes can be posed as a labelling problem, where one tries to label each vertex of a given query shape $\mathcal{X}$ with the index of a corresponding point on some reference shape $\mathcal{Y}$ [42]. We can deal this problem with the similar way when we do in the last problem, graph node classification on Cora citation network. For a point i in the query shape $\mathcal{X}$, the output of the last layer of the network produces m-dimensional vector denoted by the probability of i mapping to m nodes on the reference shape $\mathcal{Y}$. Each node on the query shape $\mathcal{X}$ has a unique registration node in the reference shape $\mathcal{Y}$. Learning is then done by minimizing soft cross entropy loss. We can use similar architecture used in graph node classification.

The difference of this problem with the previous one is the underlying structure is manifolds or discretize 3D meshes for usage. Bronstein et al. [6] point out that intrinsic convolution operators are more capable of dealing such problems than spectral convolution operators. We apply GCN, ChebyConv and B-SplineConv on this problem to verify this property.

We use the FAUST dataset [4], containing 10 scanned human shapes in 10 different poses, resulting in a total of 100 non-watertight meshes with $6,890$ nodes each. The first 80 subjects in FAUST were used for training and the remaining 20 subjects for testing. Ground truth correspondence for each node in each query shape $\mathcal{X}$ are given in the exact same order (*i.e.* vertex i in the query shape is labeled as i).

Architecture

Since the dataset is more complicated than the last problem, we use the below architecture for all the operators: Input $\rightarrow$ BasicModule[32] $\rightarrow$ 4 * BasicModule[64] $\rightarrow$ Lin(256) $\rightarrow$ Lin(6990). BasicModule denotes either of GCN, ChebyConv, B-SplineConv or FC. Lin(o) denotes a 1×1 convolutional layer to o output featues per node. As non-linear activation function, we apply ReLU at after each convolutional layer and the first Lin layer. For the last layer of the network, we apply log-softmax function in order to evaluate the loss with cross entropy loss. Training is done for all the experiments for 100 epochs with a batch size of 1. Adam optimizer is used with the initial learning rate 0.01. Table 4.6 reports the description of each basic module and the corresponding hyperparameter setting we used.

Method	Filter	Hyperparameters
GCN (Eq. 4.20)	$\alpha\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{W}}\tilde{\mathbf{D}}^{-1/2}\mathbf{f}_l$	No
ChebyConv (Eq. 4.16)	$\sum_{j=0}^{r-1}\alpha_j T_j(\hat{\Lambda})\mathbf{\Phi}^{\mathsf{T}}\mathbf{f}_l$	$r=2$
B-SplineConv (Eq. 4.35)	$\sum_{j\in\mathcal{N}(i)} f_l(i)\sum_{\mathbf{p}\in\mathcal{P}}\mathcal{W}_{\mathbf{p},l,l'}\cdot\prod_{i=1}^{d}N_{i,p_i}^{m}(u_i)$	$m=1,k=5$
FC	$\mathbf{F}_l\cdot\mathbf{W}_{l,l'}$	No

Table 4.6: Description of the basic module used in this problem.

Results

We report the prediction accuracy of each node on meshes in the testset. Figure 4.12 shows the prediction accuracy of each node compared with the groundtruth label of each node. It indicates that both ChebyConv and B-SplineConv can learn meaningful information from 3D shapes, while it's harder for GCN to extract underlying feature and we cannot see any fluctuation as for FC. B-SplineConv achieved the highest prediction accuracy around 95.7%, which attributed to its stronger capacity of extracting intrinsic shape context. Therefore, we would focus on B-SplineConv when we evaluate the problem of 4D facial expression analysis because its underlying structure is also manifold.

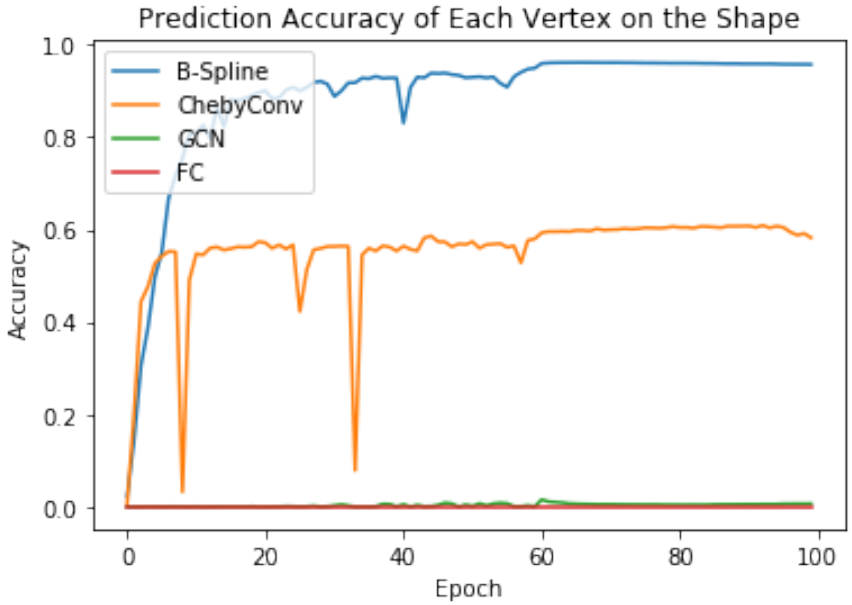


Figure 4.12: Prediction accuracy of each node of the meshes in testset.

Chapter 5

Protein Function Prediction

In the last a few years, an abundance of large-scale protein function networks was produced by high-throughput experimental methods. The connectivity of these networks provides a rich source of information for inferring functional annotations for genes and proteins. An important challenge has been to develop methods to extract useful feature representations for function prediction. With the appearance of geometric deep learning [6], which shows advantageous in capturing underlying features in graph- or manifold-structured data. Especially in Section 4.3.1, we already show that geometric deep learning methods achieve high prediction accuracy over the task, *semi-supervised multi-class classification* on Cora citation network [53]. We therefore motivated to apply geometric deep learning methods on this problem. However, **we highlight the key difference between these two problems, leading protein function prediction to be a harder and different problem compared to citation network prediction**: it is necessary to point out the key difference between these problems:

1. Only topological information (see Figure 5.2, human protein-protein interaction network) provided in protein function prediction. Cora citation network provides $1,433$ dimensional sparse binary bag-of-words feature vector describing the content of each node (*i.e.* publication), while **no relevant information** is provided for each node (*i.e.* protein or gene) in protein/gene interaction network.

2. Unbalanced multi-label classification. As for Cora citation network prediction, each node is classified to 1 over 7 classes by minimizing cross entropy loss over the probability of the true label. However, in protein function prediction this problem, each protein should be classified to **multiple labels** (*i.e.* functional annotations, MF, BP, CC). We cannot achieve this goal by minimizing cross entropy loss. It should also be noted that the amount of functional annotations for each protein is varied. Then, it is not allowed to solve the problem from finding the best k prediction classes.

3. Large sparse state space. We take the example of human MF functional annotations for illustration. The state space for each protein is 274 and the total state space is $4,377,972$. The positive states account for only 0.571%, where if protein $\mathcal{X}$ contains function $\mathcal{Z}$, the state of this protein is labeled as positive.

4. Extremely high dimensional topology. Human BioGRID network contains $1,682,361$

edges and $15,087$ nodes, while Cora citation network only has $5,278$ edges and $2,708$ nodes.

In the following sections in this chapter, we first give a concise definition of protein function prediction this problem with computer science terminology. Then we show our efforts on this problem at the beginning and problems we met at that stage. Finally, we show how we solve this problem and achieve the current state-of-the-art performance over this problem.

5.1 Problem Definition

Given multiple protein interaction networks (*e.g.* human, mouse, yeast BIOGRID or BioGRID protein interaction networks) and functional annotations (molecular function (MF), biological function (BP), cellular function (CC)) for partial proteins, the task is to predict the functional annotations for the remaining proteins. In order to deal the problem with geometric deep learning techniques, we model data as below:

Data The main attributes of data contain:

- For BIOGRID dataset, we construct an adjacency matrix $\mathcal{A} \in \mathbb{R}^{N \times N}$, where N represents N nodes. If there is an edge $e = (i, j)$ connecting nodes i and j, the entry $\mathcal{A}_{ij} = 1$; otherwise, $\mathcal{A}_{ij} = 0$. For BioGRID dataset, we construct a weighted adjacency matrix $\mathcal{W} \in \mathbb{R}^{N \times N}$ because the strengths of associations between proteins are provided (in range $[0, 1]$).

- Functional annotation (molecular function (MF), biological function (BP), cellular function (CC)) matrix $\mathbf{Y} \in \mathbb{R}^{N \times C}$, where C represents C distinctive functional annotations.

- Pseudo-coordinates $\mathcal{U} \in \mathbb{R}$. Pseudo-coordinates $u(i, j)$ is defined as globally normalized degree of target nodes i (Eq. 5.1), where $j \in \mathcal{N}(i)$ and $\mathcal{N}(i)$ denotes neighbors of i. $\mathcal{V}$ denotes a finite set of nodes in the graph.

$$u(i, j) = \frac{deg(j)}{\max_{v \in \mathcal{V}} deg(v)} \tag{5.1}$$

5.2 Multi-Layer Graph Convolution Network

Inspired by experiments on Cora citation network (Sec. 4.3.1), we propose to build a multi-layer graph convolution network based on GCN [34], ChebyConv [15] and B-SplineConv [18]. To the best of our knowledge, this is the first mehod that apply geometric deep learning technique to learn and inference a protein-protein interaction network. We show our approach, experiments and results in the following sections. In the end, we indicate issues we meet at this stage.

5.2.1 Approach

In this section, we introduce three methods, architectures, measures of how we assess prediction performance, and data preprocessing procedure.

Notations and Definitions

We define protein-protein interaction network as a connected, undirected adjacency graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$, which consists of a finite set of vertices $\mathcal{V}$ with $|\mathcal{V}| = N$, a set of edges $\mathcal{E}$, and an adjacency weighted matrix $\mathbf{W} \in \mathbb{R}^{N \times N}$. If there is an edge $e = (i, j)$ connecting vertices i and j, the entry $\mathbf{W}_{ij} = 1$ (or weight w_{ij} for BioGRID network); otherwise, $\mathbf{W}_{ij} = 0$.

The *non-normalzied graph Laplacian* is $\mathbf{\Delta} = \mathbf{D} - \mathbf{W}$, where the *degree matrix* $\mathbf{D} = \mathrm{diag}(\sum_j \mathbf{W}_{ij})$, and *normalized definition* is $\widetilde{\mathbf{\Delta}} = \mathbf{D}^{-1/2}\mathbf{\Delta}\mathbf{D}^{-1/2} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$. Because the graph Laplacian $\mathbf{\Delta}$ is a real symmetric matrix, it has a complete set of orthogonal eigenvecotrs, which we denote by $\mathbf{\Phi} = (\phi_1, \ldots, \phi_n)$[1], known as the *graph Fourier basis*. These eigenvectors have associated real, non-negative eigenvalues $\lambda_1, \ldots, \lambda_n$, identified as the frequencies of the graph, satisfying $\mathbf{\Delta}\phi_i = \lambda_i\phi_i$, for $i = 1, 2, \ldots, n$. The *graph Fourier transform* of a signal $\mathbf{F} \in \mathbb{R}^n$ is then defined as $\hat{\mathbf{f}} = \mathbf{\Phi}^\mathsf{T}\mathbf{f}$, and its inverse as $\mathbf{f} = \mathbf{\Phi}\,\hat{\mathbf{f}}$.

Architecture

We propose a multi-layer graph convolution architecture shown in Figure 5.1. We compare the performance of models based different geometric deep learning methods. In order to make the comparison fair, we compared three methods using the same two convolution layers architectures, interleaving with dropout to improve robustness of model extracting useful features. Additionally, we set an architecture with 6 splineconv layers and 2 fully connected layers to explore the improvement of performance after increasing model complexity. During forward process, the node feature matrix is updating from layer to layer and at the last layer, neural network outputs a C-dimensional vector for each node, representing the prediction probability (range in $[0, 1]$) in terms of C classes. Learning is done by minimizing binary cross entropy loss. We show details for each block in Figure 5.1.

Input With original graph $\mathcal{G}$, We obtain adjacency matrix $\mathcal{A} \in \mathbb{R}^{N \times N}$ and pseudo-coordinate $\forall i, j \in \mathcal{N}(i),\ u(i, j) = deg(j)/\max_{v \in \mathcal{V}} deg(v)$. We trivially define node feature matrix $\mathbf{F} = [1, \ldots, 1]^\mathsf{T} \in \mathbb{R}^{N \times 1}$. During forward process, the model update node feature matrix. For networks based on GCN and ChebyConv, $\mathcal{A}$ and $\mathbf{F}$ are used as input. For network based on B-SplineConv, additional pseudo-cooridinates are used.

Dropout During training, randomly zeroes some of the elements of the input node feature matrix $\mathbf{F}$ with probability p ($p = 0.5$ in all our experiments) using samples from a Bernoulli distribution. This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper [27].

Furthermore, the outputs are scaled by a factor of $1p$ during training, which means during evaluation the module simply computes an identity function.

[1]Note that there is not necessarily a unique set of graph Laplacian eigenvectors, but we assume throughout that a set of eigenvectors is chosen and fixed.

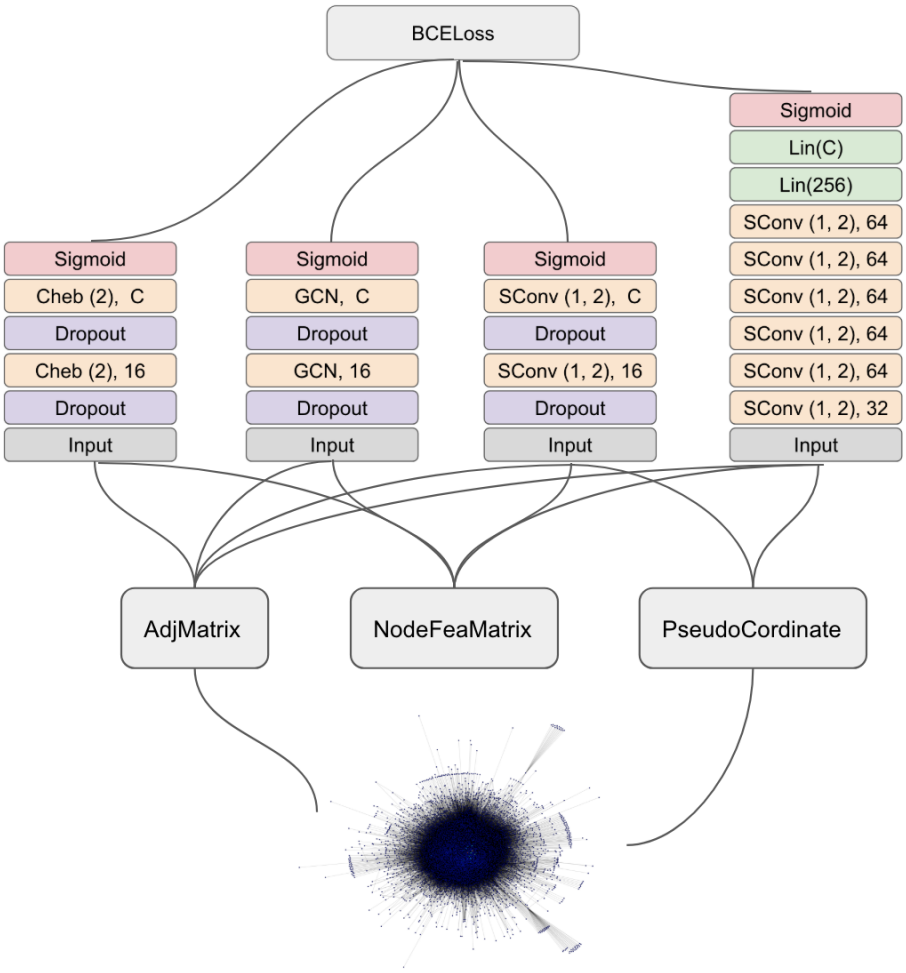


Figure 5.1: Architectures employed for our experiments. ReLU activation function is used for the left two architectures, whereas, ELU activation function is used for the right two architectures. For all architectures, binary cross entropy loss is used. To make the comparison between Cheb and GCN fairly, Cheb used the degree of 1 (so with $r = 2$). The kernel dimension for SConv is 1, and kernel size is 2. Adjacency matrix $\mathcal{A}$ and pseudo-coordinates are obtained from protein interaction network. We trivially initialize node feature matrix $\mathbf{F} = [1, \ldots, 1]^\mathsf{T} \in \mathbb{R}^{N \times 1}$.

Convolution We assume the input node feature for k_1 layer is $\mathbf{F}^{(k_1)} \in \mathbb{R}^{N \times C^{k_1}}$ and the output node feature is $\mathbf{G}^{(k_2)} \in \mathbb{R}^{N \times C^{k_2}}$. We denote $\mathbf{f}_l$ and $\mathbf{g}_{l'}$ as l^{th} and l'^{th} column of $\mathbf{F}^{(k_1)}$ and $\mathbf{G}^{(k_2)}$ respectively.

- *ChebyConv*. We employ ChebyConv as below to update feature in terms of each node:

$$\mathbf{g}_{l'} = \sum_{l=1}^{C^{k_1}} \left(\sum_{j=0}^{r-1} \alpha_j T_j(\hat{\Delta}) \mathbf{f}_l + \mathbf{b}_{l,l'} \right) \quad l' = 1, \ldots, C^{k_2}$$

  where $\mathbf{b}_{l,l'}, \alpha_j, T_j$ denotes the bias, the learnable parameters and the chebyshev polynomial of degree j, respectively.

- *GCN*. We update node feature information from the following formula:

$$\mathbf{g}_{l'} = \sum_{l=1}^{C^{k_1}} \alpha \tilde{\mathbf{D}}^{-1/2} \tilde{\mathbf{W}} \tilde{\mathbf{D}}^{-1/2} \mathbf{f}_l + \mathbf{b}_{l,l'} \quad l' = 1, \ldots, C^{k_2}$$

  where $\tilde{\mathbf{W}} = \mathbf{W} + I$ and $\tilde{\mathbf{D}} = \sum_j \tilde{W}_{ij}$. $\mathbf{b}_{l,l'}$ represents the bias. α is learnable parameter.

- *BSplineConv*. We use the following equation to aggregate feature of node i from neighbor nodes $j \in \mathcal{N}(i)$:

$$\mathbf{g}_{l'}(i) = \frac{1}{|\mathcal{N}(i)|} \sum_{l=1}^{C^{k_1}} \left(\sum_{j \in \mathcal{N}(i)} \mathbf{f}_l(i) \sum_{\mathbf{p} \in \mathcal{P}(\mathbf{u}(i,j))} \mathcal{W}_{\mathbf{p},l,l'} \cdot \prod_{k=1}^{d} N_{k,p_i}^m(u_k) + b_{l,l'} \right) \quad l' = 1, \ldots, C^{k_2}$$

  where N_{k,p_i}^m and $\mathcal{W}_{\mathbf{p},l,l'}$ denotes B-spline basis over degree m and learnable parameters, respectively.

Nonlinear Activation Function We apply element-wise nonlinear activation function $\xi(x)$ at the output of each convolution layer. As for models based on ChebyConv and GCN, we apply ReLU activation function, while ELU is used for models based on BSplineConv. At the output of the last layer, all models are applied sigmoid function to enforce the network to output values ranging $(0, 1)$, representing probabilities over different functional annotations.

$$\xi(x) = \text{Sigmoid}(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

Loss Function In order to force the network to learn the transformation from mapping node i to multiple labels, we use binary cross entropy loss to perform this, which works well especially in the case where each network output is treated independently and labels are represented as binary values 0 or 1. We assume the network output is matrix $\mathbf{X} \in \mathbb{R}^{N \times C}$ and the target is matrix $\mathbf{Y} \in \mathbb{R}^{N \times C}$, we minimize the following loss $\mathcal{L}$:

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{NC} \sum_{i=1}^{N} \sum_{j=1}^{C} \ell_{ij} \tag{5.2}$$

$$\ell_{ij} = -w_{ij}[y_{ij} \cdot \log x_{ij} + (1 - y_{ij}) \cdot \log(1 - x_{ij})] \tag{5.3}$$

where w_{ij} denotes the manual rescaling weight given to the loss of each element.

5.2.2 Assessment of Performance

We model the problem of protein function as a multilabel classification problem. The adjacency matrix $\mathcal{A} \in \mathbb{R}^{N \times N}$ and node feature matrix $\mathbf{F} \in \mathbb{R}^{N \times 1}$, which are used as input for neural network models. We aim to produce score matrix with the same shape to matrix of label matrix, *i.e.* $\mathbf{S} \in \mathbb{R}^{N \times C}$ and $\mathbf{Y} \in \mathbb{R}^{N \times C}$ respectively. In practice, it should be noticed

that there are comparable amounts of proteins not annotated, which should be filtered in advance, leading to input node matrix as $\mathbf{X} \in \mathbf{R}^{N^* \times 1}$, where $N^* < N$. We randomly split all annotated proteins into a training set, comprising 80% of annotated proteins, and a testset, comprising the remaining 20% of annotated proteins. The performance is averaged over 5 times repeated experiments.

For each method, we use the following metrics to evaluate the prediction performance: (i) *Micro-averaged F1 score (F1)* is computed in the same way as in Cho et al. [11] (2016); (ii) *Micro-averaged area under the precision-recall curve (m-AUPR)* is computed by first vectorizing the proteinfunction matrices of predicted scores and known binary annotations, and then computing the AUPR by using these two vectors; *Macro-AUPR (M-AUPR)* is computed by first computing the AUPR for each function separately, and then averaging these values across all functions.

5.2.3 Data Preprocessing

We report the results of the methods on human with 15,978 nodes and 217,076 edges (Figure 5.2), mouse with 5,440 nodes and 13,250 edges (Figure 5.3) and yeast with 5,932 nodes and 88,677 edges (Figure 5.4) BioGRID networks. All functional annotations are taken from Gene Ontology (GO), containing molecular function (MF), biological process (BP) and cellular component (CC) GO terms. To make the model performance comparable to *deepNF* [19] and *Mashup* [11], human MF annotations are explicitly arranged into two functional categories, *i.e.* categories containing GO terms annotating 31-100 (covering 194 MF GO terms) and 101-300 (covering 80 MF GO terms) proteins, respectively.

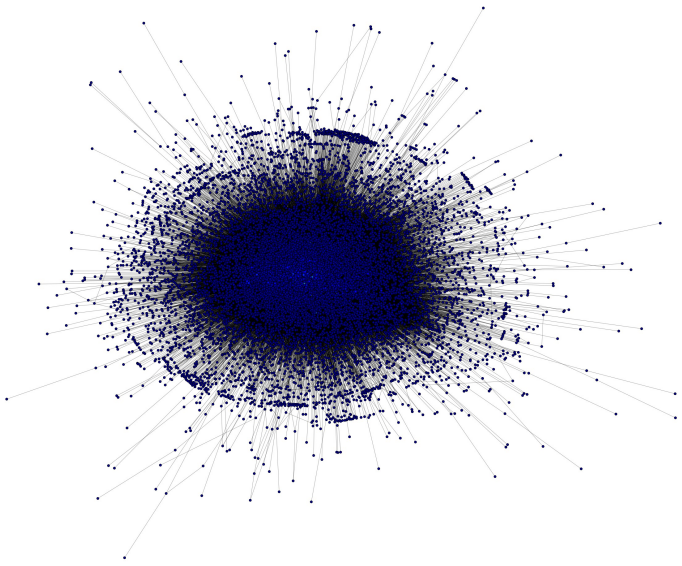


Figure 5.2: Human BioGRID network with 15,978 nodes and 217,076 edges. Each node is colored as the sort of node degree from the highest to the lowest, yielding shallow color to dark blue color.

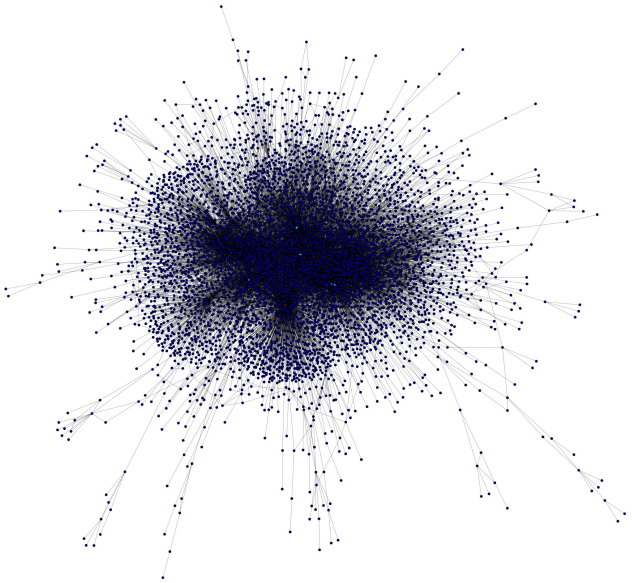


Figure 5.3: Mouse BioGRID network with 5,440 nodes and 13,250 edges. Each node is colored as the sort of node degree from the highest to the lowest, yielding shallow color to dark blue color.

5.2.4 Results

We perform experiments with four architectures on three distinct BioGRID networks, namely Human, Mouse and Yeast, using MF, BP, CC functional annotations respectively. All experiments are trained with NVIDIA GTX 1080 for 200 epochs, using Adam optimizer with the initial learning rate 0.01, divided by 10 when loss is in plateaus after 20 epochs, and dropout probability 0.5. All the hyperparameter setting for ChebyConv and BSplineConv are taken from the Cora experiments results reported in Section 4.3.1. As for loss function, binary cross entropy loss (Eq. 5.2) is used after the network's sigmoid output. We stress that the **loss should be initialized to give a higher weight to positive functional annotations** because labels are highly unbalanced (*e.g.* positive annotations accounts for only 0.571% of the total human MF annotations). Basically for all experiments, we initialize the weight matrix with the value 100 for positive annotations and 1 for negative counterparts, whereas, this setting is still required to further explore.

Human BrioGRID network

The performance of three methods (four architectures) applied on Human SRING network is shown in Figure 5.5. As for 2 layers architecture, Cheb-based architecture significantly outperforms the other two methods, in terms of all three measures. In particular, **Cheb-based architecture shows better M-aupr performance than 6-layers Spline-based architecture**, for the MF-GO terms belonging to the most specific (*i.e.* annotating between

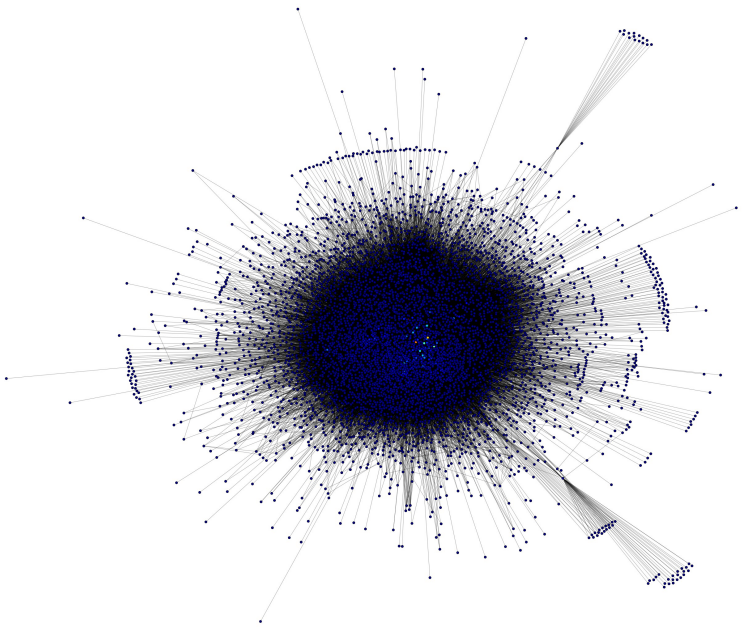


Figure 5.4: Yeast BioGRID network with 5,932 nodes and 88,677 edges. Each node is colored as the sort of node degree from the highest to the lowest, yielding shallow color to dark blue color.

31 and 100 proteins) categories. Generally, for the MF-GO terms belonging to both general and specific (*i.e.* annotating between 31 and 300) categories, we observe higher performance of 6-layer Spline-based architecture, except in terms of M-aupr performance, for which 6-layer Spline-based architecture is comparable with 2-layer Cheb-based architecture. Moreover, 6-layer Spline-based architecture has the significantly better performance compared to its 2-layer architecture, which **proves the effectiveness of deeper neural networks on this problem**.

Similar results (Figure 5.6) are observed for BP ontologies, where 6-layer SplineConv and 2-layer show comparable performance in terms of measures such as M-aupr and m-aupr, while F1 score of 6-layer architecture is better than 2-layer Cheb-based architecture. By comparison, 6-layer Spline-based architecture proves to be the best with respect to predict human CC functional annotations. It should be noticed that the rate of positive examples of BP is smaller than MF and CC annotations (0.546% vs 0.571% and 0.572%), which leads to lower scores on all of three measurements.

Mouse and Yeast BioGRID networks

From the results of predicting Mouse MF, BP, CC annotations (Figure 5.7), we observe that the **performance improvement from deeper model is disappear when dealing with BioGRID networks with lower complexity** (mouse BioGRID network containing 5,440 nodes and 13,250 edges, compared to human's 15978 nodes and 217,076 edges). 2-layer or 6-layer Spline-base architectures and 2-layer Cheb-based architecture shows similar performance on all MF, BP, CC annotations, and better than GCN-based architecture.

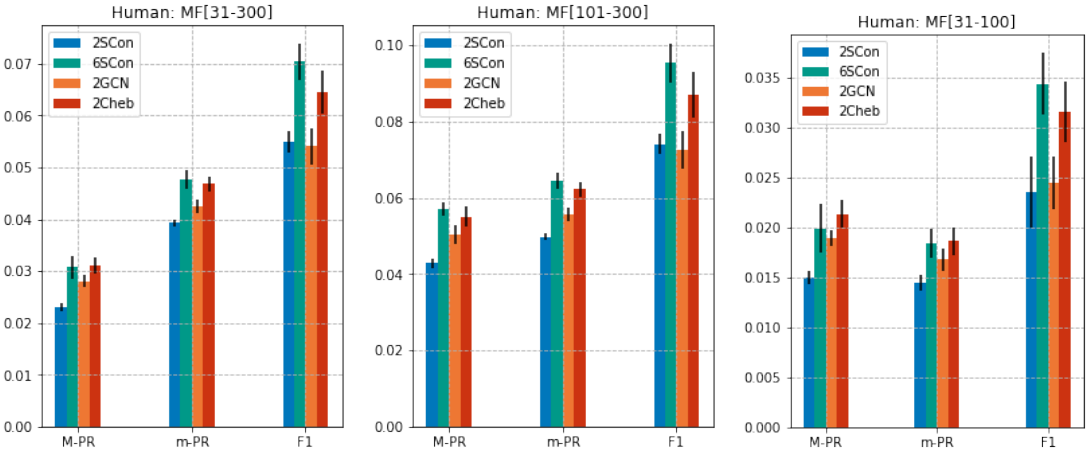


Figure 5.5: The performance of three methods (four architectures) in analyzing human BioGRID networks with MF functional annotations (274 annotations in total), which in particular MF ontology (from 31 to 300) is further divided into two levels annotating 101-300 and 31-100 proteins respectively. Performance is measured by the area under the precision-recall curve, summarized over all GO terms both under the micro-averaging (m-AUPR), macro-averaging (M-AUPR) schemes and F1 score. The error bars are computed based on 5 trials.

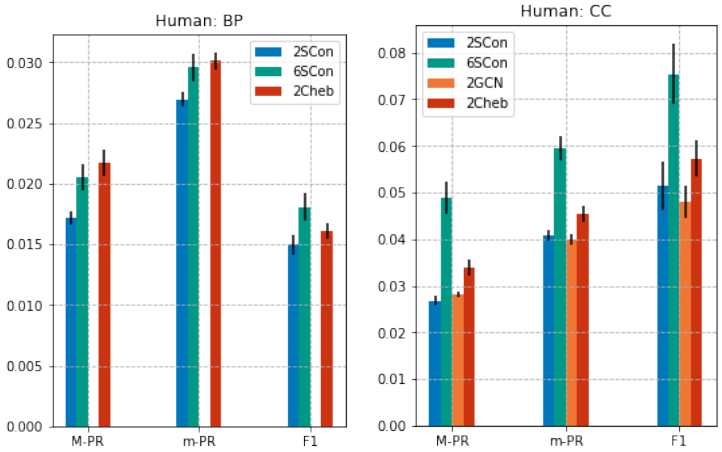


Figure 5.6: The performance of three methods (four architectures) in analyzing human BioGRID networks with BP (1282 annotations in total) and CC (225 annotations) functional annotations. Performance is measured by the area under the precision-recall curve, summarized over all GO terms both under the micro-averaging (m-AUPR), macro-averaging (M-AUPR) schemes and F1 score. The error bars are computed based on 5 trials. Because of high GPU memory usage of GCN with respect to BP annotations (more than 8GB) and our GPB resource is not satisfied, we didn't get the result of this series experiments.

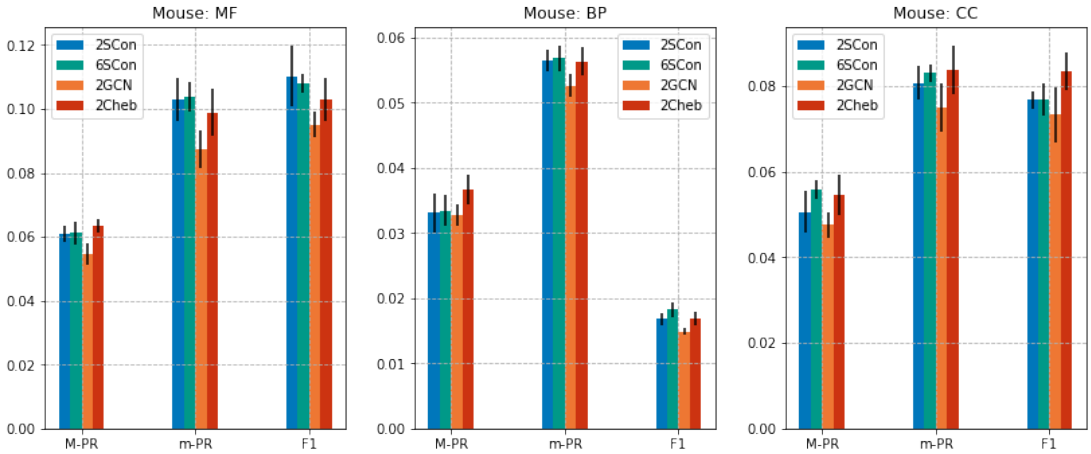


Figure 5.7: The performance of three methods (four architectures) in analyzing mouse BioGRID networks with MF (117 annotations), BP (1077 annotations) and CC (157 annotations) functional annotations. Performance is measured by the area under the precision-recall curve, summarized over all GO terms both under the micro-averaging (m-AUPR), macro-averaging (M-AUPR) schemes and F1 score. The error bars are computed based on 5 trials.

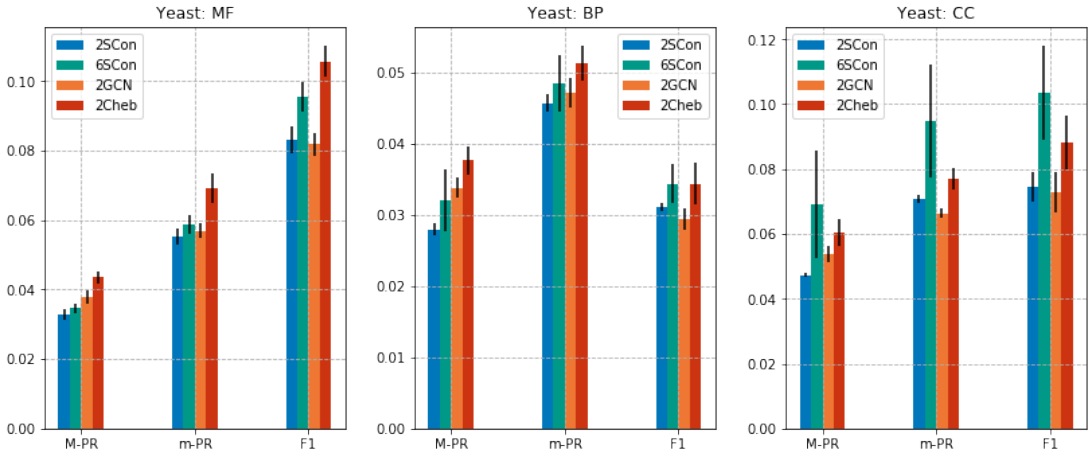


Figure 5.8: The performance of three methods (four architectures) in analyzing yeast BioGRID networks with MF (157 annotations), BP (688 annotations), CC (170 annotations) functional annotations. Performance is measured by the area under the precision-recall curve, summarized over all GO terms both under the micro-averaging (m-AUPR), macro-averaging (M-AUPR) schemes and F1 score. The error bars are computed based on 5 trials.

A different trend shown on the results of Yeast (Figure 5.8) in terms of MF, BP, CC annotations. One distinctive observation is that Cheb-based architecture significantly outperforms other methods in terms of MF and BP annotations. **We attribute this phenomenon to ChebConv's higher quality of features extraction from the complex topology of BioGRID networks**, since the same order of nodes between Yeast (5932) and Mouse (5440), but more edges of Yeast (88,677) than Mouse (13,250). Additionally, **with higher complexity of BioGRID networks (yeast compared to mouse), deeper models shows advantageous again on all MF, BP, CC functional annotations**.

5.2.5 Discussion

From results and analysis in the last section, we can make the following conclusions: (i) Chebyshev convolution outperforms other methods (B-Spline basis convolution and GCN) with respect to extracting protein features from BioGRID networks, and the greater performace difference is observed if the complexity of the topological structure increases; (ii) deeper model is considered to show better prediction performance in terms of human and feast BioGRID networks, but this superiority is disappeared for mouse BioGRID networks, which is attributed to lower complexity of BioGRID networks. It's also worth indicating that the training time for Chebyshev-based architecture is the smallest (Table 5.1).

	2SConv	6SCon	2GCN	2Cheb
Training Time (1 epoch)	0.28s	**0.92**s	0.35s	**0.15**s

Table 5.1: The training time for each architecture in 1 epoch. All experiments are performed on the same NVIDIA GTX 1080.

How to evaluate the results?

As functional annotations (human and yeast, MF, BP, CC annotations) used in existing publications (deepNF [19], Mashup [11], GeneMANIA [45]) are not the same with our experiments, it's hard to compare results fairly with other existing methods, whereas, our results are more likely worse than others (deepNF, Mashup, GeneMANIA) because one order of magnitude of performance can be observed. Generally, for any functional annotation, if it is correctly predicted in terms of all proteins (*i.e.* all positive annotation given higher scores than negative annotations), M-AUPR result for this function then should be higher than 0.97 (take example of human MF annotations). If we hold the assumption that all the methods we chose are capable of extracting useful features representations lying in graph-structured data, we then should do some further exploration to find where issues come from.

Analysis

The trends of train and test loss provide more information than accuracy (M-AUPR, m-AUPR, F1 scores here). Figure 5.9 show all the four architectures are already converged

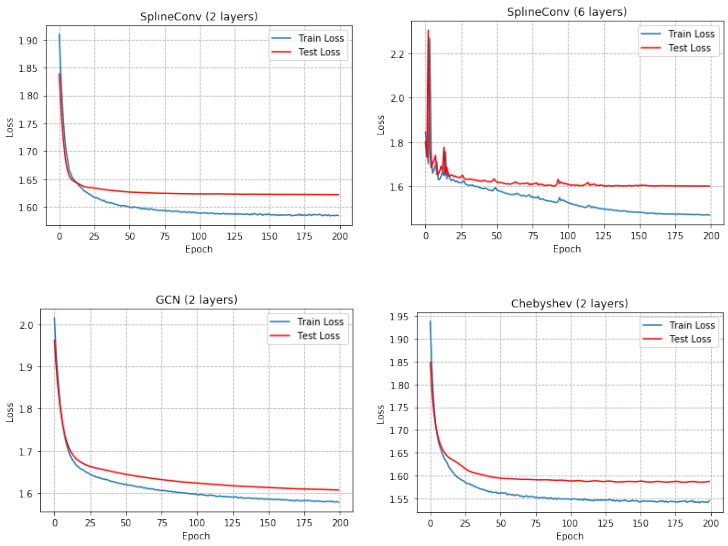


Figure 5.9: The loss of three methods (four architectures) in analyzing human BioGRID networks with MF functional annotations. Both train and test loss are averaged over 5 trials in 200 epochs. The initial learning rate is 0.01 and decayed when test loss is in plateaus in 20 epochs.

after 200 epochs, and since learning rate is decaying after 20 epochs in plateaus, it's unlikely that learning rate is not small enough. Another possibility leading might be wrong weight initialization, leading the models to fall into local minima every time. We then evaluated 2Cheb-based architecture with *Kaiming Initialization* [24], *Xavier Initialization* [20], and additionally we ran experiments in 1000 epochs with initial learning rate 0.01, divided by 10 every 100 epochs. Moreover, we keep tracking of the changing of gradient by integrating gradients in different layers and computing the frobenius norm. From Figure 5.10, we observe Kaiming and Xavier can accelerate convergence, yielding a slightly lower training loss but not remarkable improvements in terms of test loss, meaning the generalization performance not changed. It's more likely that the model is actually converged to a flat minimizer, which is generally the suggested point for training deep neural networks.

At this stage, we suspect that issues may come from BCELoss function since the output state space is extremely large (millions) and sparse, which capability may be not enough to achieve such multi-label classification task. We have to find better strategy to encourage true positive and punish false negative outputs.

5.3 Denoising Graph Autoencoder with SVM Classifer

Inspired by the work deepNF (Gligorijević et al. [19]) on this problem , we are then motivated to design a *graph autoencoder architecture*, which is capable of learning a compact, low-dimensional latent feature presentation from graph-structured data in a fully unsupervised way and more importantly it is independent of the function prediction task. This allows for the use of the entire dataset in the training of the graph autoencoder, resulting

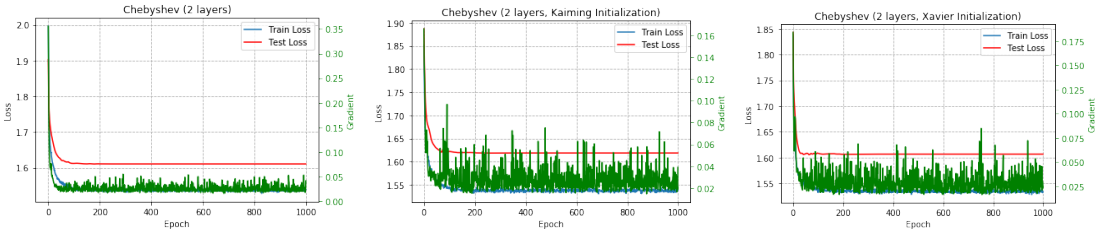


Figure 5.10: The loss of three methods (four architectures) in analyzing human BioGRID networks with MF functional annotations. Both train and test loss are averaged over 5 trials in 1000 epochs. The initial learning rate is 0.01, divided by 10 every 100 epochs. The left comparison experiment keeps the original weight setting, and the middle and the right one are initialized by Xavier and Kaiming initialization respectively.

in high-quality features. Additionally, we would incorparate the idea of denoising autoencoder introduced in Sec. 2.2.2, which proved to produce the higher level representations (stable and robust). In order to enable semi-supervised multi-label classificaiton, the extracted features are then used to train a SVM. In Sec. 4.3.1, we have already proved that geometric deep learning methods (especially GCN, ChebConv, BSplineConv) perform far more better than stacking fully connected layer since it cannot take use of additional topology information during propagation. We therefore believe we could undoubtly achieve better performance than deepNF, which provides the current state-of-the-art performence on this problem.

In the following subsections, we would show how we build a graph autoencdoer to encode latent graph feature representation in an unsupervised way, and then show our experimetns results.

5.3.1 Approach

In this section, we use *Notations and Definitions* defined in the Section 5.2.1.

Architecture

The Figure 5.11 shows the architecture of our model. We start from training the denoising graph autoencoder with full dataset until the model converges. Then we extract the latent feature representation from the bottleneck layer and train a SVM classifer to predict multiple functional annotations in terms of each protein.

We denote the input node feature matrix (left, Figure 5.11) $\mathbf{F}_{in} = [1, \ldots, 1]^{\mathsf{T}} \in \mathbb{R}^{N \times 1}$. The output of the autoencoder is $\mathbf{F}_{out} \in \mathbb{R}^{N \times 1}$.

Corruption Process We consider a common noise model, isotropic *Gaussian noise* (GS). Therefore, the corrupted node feature matrix $\tilde{\mathbf{F}}|\mathbf{F} \sim \mathcal{N}(\mathbf{F}, \sigma^2 I)$. This corrupted matrix is then fed to the aotoencoder training.

Encoder During forward process, the graph topology is fixed and the same adjacency matrix $\mathcal{A}$ is used during chebyshev convolution. We stack multi-layer chebyshev convo-

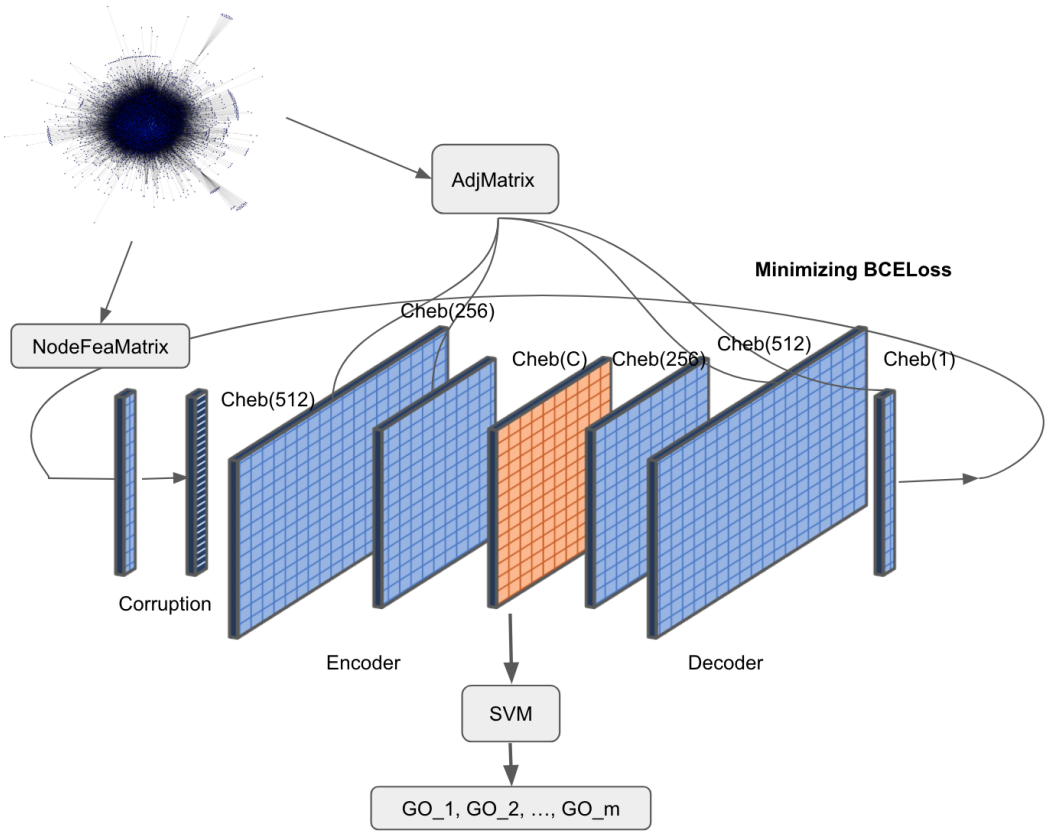


Figure 5.11: The architecture of denoising graph autoencoder incorporated with SVM classifier.

lution to update node feature information so as to capture high-dimensional topological structure lying in graphs. Basically, at each layer of the encoder, we update node feature using the following equation:

$$\mathbf{g}_{l'} = \sum_{l=1}^{C^{k_1}} \left(\sum_{j=0}^{r-1} \alpha_j T_j(\hat{\Delta}) \mathbf{f}_l + \mathbf{b}_{l,l'} \right) \quad l' = 1, \ldots, C^{k_2}$$

where $\mathbf{b}_{l,l'}, \alpha_j, T_j$ denotes the bias, the learnable parameters and the chebyshev polynomial of degree j, respectively.

We assume the output is $\mathbf{G} \in \mathbb{R}^{N \times C^{k_2}}$. We take the idea of batchnormalization [29], but now we would normalize the whole node feature matrix over each channel, which proved to be able to accelerate deep network training by reducing internal covariate shift.

$$\mathbf{g}'_{\mathbf{l}} = \frac{\mathbf{g}_l - \mathbf{E}(\mathbf{g}_l)}{\sqrt{\mathbf{Var}(\mathbf{g}_l) + \epsilon}} \cdot \gamma + \beta$$

where $\mathbf{g}_l \in \mathbb{R}^{N \times 1}, \mathbf{E}(\mathbf{g}_l), \mathbf{Var}(\mathbf{g}_l)$ denotes the l^{th} column of the matrix $\mathbf{G}$, mean and variance of $\mathbf{g}_l$, respectively. γ and β represent learnable parameters. During training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation.

Instead of general choice of ReLU, we use a *Parametric Rectified Linear Unit* (PReLU) [24] introduced in Section 2.1.3. Formally, it is defined as the equation 5.4. Here x is the input of the nonlinear activation ξ operating on each entry of node feature matrix $\mathbf{F}$. a is a coefficient controlling the slope of the negative part. If a is a small and fixed value, PReLU becomes the LeakyReLU introduced in Sec. 2.1.3. The motivation of LReLU is to avoid zero gradients. However, experiments in [41] show that LReLU has negligible impact on accuracy compared with ReLU. On the contrary, He et al. [24] show PReLU adaptively learns the parameters jointly with the whole model presenting superiority. They indicate that from introducing a very small number of extra parameters, PReLU activation gradually become "more nonlinear" at increasing depths. In other words, the learned model tends to keep more information in earlier stages and becomes more discriminative in deeper stages.

$$\xi(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases} \tag{5.4}$$

In the last layer of encoder, we use sigmoid activation function to enforce the outputs ranging in $(0, 1)$.

Decoder The aim of the decoder is to map the latent feature matrix $\mathbf{H} \in \mathbb{R}^{N \times C}$ to the matrix with the shape with the input node feature matrix, *i.e.* $\mathbf{F}_{out} \in \mathbb{R}^{N \times 1}$. We take the same methodologies as the encoder but keep increasing the dimension of the node feature matrix. It should be noticed that in the last layer of the decoder when mapping $\mathbf{F} \in \mathbb{R}^{N \times C_{k_2}}$ to $\mathbf{F}_{out} \in \mathbb{R}^{N \times 1}$. We apply sigmoid activation function at the output, which then allows us to minimize the model's input and output with binary cross entropy loss.

Loss Function At each epoch, learning is done from minimizing the following equation between initial input (not corruption one $\tilde{\mathbf{F}}$) $\mathbf{F}_{in}$ and $\mathbf{F}_{out}$:

$$\mathcal{L}(F_{in}, F_{out}) = \frac{1}{N} \sum_{i=1}^{N} \ell_i \tag{5.5}$$

$$\ell_i = y_i \cdot \log(x_i) + (1 - y_i) \cdot \log(1 - x_i) \tag{5.6}$$

where x_i and y_i denotes the i^{th} element of the vector $\mathbf{F}_{in}$ and $\mathbf{F}_{out}$, respectively.

SVM Classifier

We use the compressed features, $\mathbf{H}$, computed in the previous step, to train an SVM classifier to predict probability scores for each protein. We use the SVM implementation provided in the LIBSVM package ([9]).

5.3.2 Assessment of Performance

To measure the performance of the SVM on the compressed features, we adopt 5-fold cross validation as our evaluation strategy.

In the *5-fold cross validation*, we split all annotated proteins into a training set, comprising 80% of annotated proteins, and a test set, comprising the remaining 20% of annotated proteins. We train the SVM on the training set and predict the function of the test proteins. We use the standard radial basis kernel (RBF) for the SVM and perform a nested 5-fold cross validation within the training set to select the optimal hyperparameters of the SVM (*i.e.* c in the RBF kernel and the weight regularization parameter, C) via *grid search*. All performance results are averaged over 10 different CV trials.

5.3.3 Data Prepossessing

We report the results of a ChebyConv-based denoising autoencoder on human BioGRID network with 15,978 nodes and 217,076 edges (Figure 5.2). All functional annotations are taken from Gene Ontology (GO), containing molecular function (MF), biological process (BP) and cellular component (CC) GO terms. To make the model performance comparable to *deepNF* [19] and *Mashup* [11], human MF annotations are explicitly arranged into two functional categories, *i.e.* categories containing GO terms annotating 31-100 (covering 194 MF GO terms) and 101-300 (covering 80 MF GO terms) proteins, respectively.

5.3.4 Results

Instead of testing the performance on all MF, BP and CC functional annotations, we only report and analyze the results on BioGRID mouse network in terms of MF functional annotation. From working on such dataset, we still can see if the issue is resolved or not, or if we can see obvious improvement brought by a popular SVM classifier.

During training *GraphAE*, We use the optimizer Adam with learning rate 0.005, divided by 5 when loss is in plateaus after 20 epochs and all the weight matrices in terms of ChebyConv are initilized with Xavier Initialization. We train the model with NVIDIA GTX 1080 for 200 epochs. Then we use the node feature matrix extracted from bottleneck layer and train a SVM classifer using the standard radial basis kernel (RBF). The SVM implementation provided in the LIBSVM package [9].

Regarding the SVM step of our method, we use the exact same grid search procedure as in the Mashup [11] paper for choosing the optimal hyperparameters:

- RBF kernel bandwidth: $\gamma \in \{0.001, 0.01, 0.1, 1.0\}$
- regularization parameter of the SVM: $C \in \{0.1, 1.0, 10.0, 100.0\}$

After training SVM with 5-fold crossvalidation via grid search for optimal hyperparameters, we report the optimal hyperpameters as following: the optimal regularization parameter of the SVM $C = 1.0$ and RBF kernel bandwidth $\gamma = 0.10$. Figure 5.12 shows the results of BioGRID mouse network over MF functional annotations. We provide the corresponding results from multi-layer ChebyConv introduced in Sec. 5.2. Figure 5.12 indicates results of GraphAE over all the metrics, *i.e. Macro-AUPR*, *Micro-AUPR* and *Micro-averaged F1 score* are lower than simply stacking graph convolution layers optimized with binary cross entropy loss. Therefore, it is not necessary to continue conduct experiments over other dataset. Moreover, we find that SVM which is suggested to work on sparse

multi-label classification cannot resolve the issue of low performance. This means the low performance is caused by other factors. In the next section, we would show how we solve this problem finally.

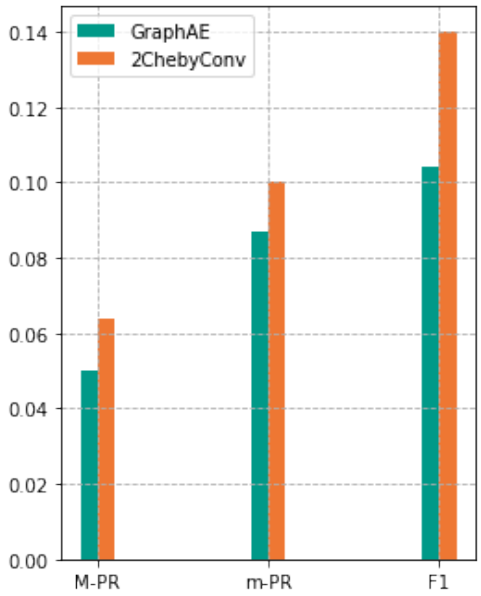


Figure 5.12: Comparison between architectures of GraphAE with SVM and Multi-Layer ChebyConv Network over BioGRID mouse network on MF functional annotations.

5.4 Novel Deep Graph Neural Networks

In the last section, we introduce how we design a graph autoencoder architecture so as to extracting graph latent feature representation. However, the relatively low performance raises the question again. Why can't we achieve good results? Since in the Section 4.3.1, we have already proved that it would undoubtedly produce better results by simply replacing a fully connected layer or 1×1 convolution to geometric deep learning methods we introduced in the Chapter 4. If we consider this as our proposition, then we should achieve better results in Sec. 5.2. In the contrast, we haven't seen results and we suspect it is because we use the wrong strategy to optimize the model. In Sec. 5.3, we turn to build a graph autoencoder so that we can take advantages of SVM classifier which is considered to work on sparse multi-label classification problem. Unfortunately, the results are still not as our expectation. After endless efforts devoted to trying different directions, we finally found that actually geometric deep learning methods are unable to capture high-dimensional topological structure directly. We need to assist the model to learn from data. It is not enough to trivially initialize node feature as one. Finally, we designed our deep graph neural networks, which achieved the current state-of-the-art performance on protein function prediction this problem. And we also argue that from taking use of our models, tasks in terms of node classification defined on general graphs can be solved without providing hand-crafted features and achieve high performance. For example, we

prove that using our models to Cora citation network, we can achieve the prediction accuracy at 87.2% which is higher than previous methods that using $1,433$-dimensional sparse bag-of-words node feature vectors.

5.4.1 Approach

Our approach mainly consist of two steps: (i) embedding high-dimensional protein interaction networks into low-dimensional vector spaces, which should preserve both the local and global network structures; (ii) Instead of initializing node feature with hand-crafted features or predefined values such as ones, we treat the embedded low-dimensional vector as node feature and train multi-layer graph convolution networks (5.2) to resolve the task of semi-supervised multi-label classification. Actually, with feature obtained from network embedding, we can also see dramatical improvements with regard to denoising graph convolutional networks (Sec. 5.3). We would show comparison experiments with regard to these two architectures to see which architecture produce better results for this problem.

Protein Interaction Network Embedding

In order to make the embedded low-dimensional vector space carrying meaningful network structural information, the high-dimensional protein interaction network embedding step should preserve the network structure. Intuitively, the local network structure, *i.e.* the local pairwise proximity between the vertices, must be preserved. This is also coincident with the inherent property of spectral and spatial convolution operators we used such GCN, ChebConv and BSplineConv, which are all designed based on 1-hop localization. However, *first-order proximity* alone is not sufficient for describing the protein interaction networks since it is highly complex and sparse. It is important to seek a network embedding method that addresses the problem of sparsity. A natural intuition is that vertices that share similar neighbors tend to be similar to each other, say *second-order proximity*. Additionally, the method should be able to deal with very large networks, say millions of edges. Therefore, we resort to the approach of Tang et al. [59] and further modify it to this problem.

For the protein interaction network, we construct high-quality vector representations of proteins, $\mathbf{F} \in \mathbb{R}^{N \times d}$, preserving potentially complex, non-linear relations among the network nodes.

First-Order Proximity The first-order proximity refers to the local pairwise proximity between the vertices in the network. To model the *first-order* proximity, we adopted the strategy proposed in [59]. For each undirected edge (i, j), the joint probability between vertex v_i and v_j is defined as below:

$$p_1(v_i, v_j) = \frac{1}{1 + \exp(-\mathbf{u}_\mathrm{i} \cdot \mathbf{u}_\mathrm{j})} \tag{5.7}$$

where $\mathbf{u}_i$ is the low dimensional vector representation of vertex v_i. To preserve the first-order proximity, we then minimize the KL-divergence,

$$\mathcal{L}_1 = -\sum_{(i,j)\in\mathcal{E}} w_{ij} \log p_1(v_i, v_j) \tag{5.8}$$

From minimizing the objective (Eq. 5.8), we can represent every vertex in the d-dimensional space.

Second-Order Proximity The second-order proximity assumes that vertices sharing many connections to other vertices are similar to each other. In this case, each vertex is also treated as a specific context and vertices with similar distributions over the contexts are assumed to be similar. Therefore, each vertex plays two roles: the vertex itself and a specific context of other vertices. Then the second-order proximity can be similarly defined as below,

$$p_2(v_j|v_i) = \frac{\exp(\mathbf{u}'^{\mathsf{T}}_j \cdot \mathbf{u}_i)}{\sum_{k=1}^{|\mathcal{V}|} \exp(\mathbf{u}'^{\mathsf{T}}_k \cdot \mathbf{u}_i)} \tag{5.9}$$

where $\mathbf{u}_i$ is the representation of v_i when it is treated as a vertex while $\mathbf{u}'_i$ is the representation of v_i when it is treated as a specific context. To preserve the second-order proximity, we then minimize the below KL-divergence,

$$\mathcal{L}_2 = -\sum_{(i,j)\in\mathcal{E}} w_{ij} \log p_2(v_j|v_i) \tag{5.10}$$

By learning $\mathbf{u}_i$ and $\mathbf{u}'_i$ that minimize Eq. 5.10, we are albe to represent every vertex v_i with a d-dimensional vector $\mathbf{u}_i$.

In practice, we trivially optimize $\mathcal{L}_1$ and $\mathcal{L}_2$ separately to enforce the embedded vector representation to preserve both the first-order and second-order proximity. Then, we then concatenate the embeddings trained by the two methdos for each vertex. In the end, we obtain the low-dimensional representation for each vertex with a d dimensional vector. We denote it in a matrix form by $\mathbf{F} \in \mathbb{R}^{N\times d}$, where N denotes $|\mathcal{V}|$ for convenience.

Semi-Supervised Multi-label Classification with Graph Convolution Networks

Now, with feature matrix $\mathbf{F}^{N\times d}$ obtained from protein interaction network embedding and the adjacency matrix $\mathcal{A} \in \mathbb{R}^{N\times N}$ obtained from the original protein interaction networks, we can conduct the task semi-supervised multi-label classification with graph convolution networks. Figure 5.13 shows an improved deep graph neural network architecture. The propagation and the strategy of updating node feature is the same with that defined in Sec. 5.2. Since in the Sec. 5.2, we already prove that ChebyConv and BSplineConv are better on this problem in terms of performance and GPU memory usage. Since the input node feature matrix is larger than we trivially defined before as $[1,\ldots,1]^{\mathsf{T}} \in \mathbb{R}^{N\times 1}$, the model training is limited by the GPU resource.

5.4.2 Results

We firstly show experiments on BioGRID mouse network in terms of MF functional annotations. We train the network with Adam optimizer with initial learning rate 0.01, divided by 10 when loss is in plateaus after 20 epochs. We intialize dropout probability 0.5. We train the same 2layer ChebyConv with Section 5.2. We train models with NVIDIA GTX

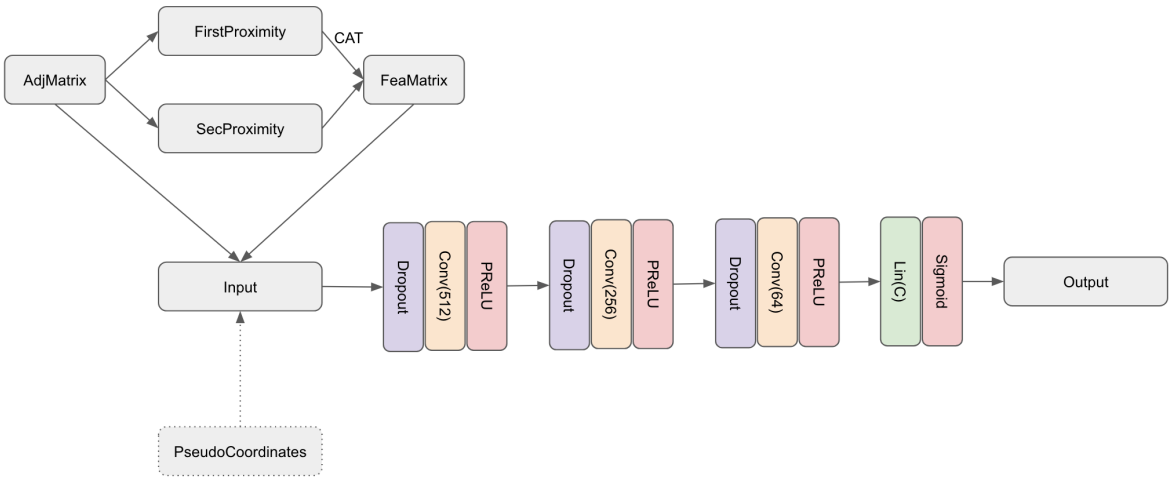


Figure 5.13: The architecture of deep graph neural network. The vertex feature obtained from minimizng first-order proximity loss and second-order proximity loss are concatenate into a node feature matrix $\mathbf{F} \in \mathbb{R}^{N \times d}$. In the last layer, network produce C-dimensional vector for each vertex. Here C denotes the classes in total. We use ChebyConv and BSplineConv as out basic convolution module. Only when using BSplineConv, the module PseudoCooridnates is required.

1080 for 200 epochs and loss function is binary cross entropy loss without initializing loss weight. We compare such network which incorporated with protein network embedding with 2layer ChebyConv network.

Comparison based on multi-layer graph convolution network The figure 5.14 shows significant improvement after we use the low-dimensional representation of protein interaction network as node feature while both models use the same multi-layer graph convolution network based on Chebyshev Convolution.

Comparison based on Denoising Graph Autoencoder Here, we compare the performance of denoising graph autoencoder with low-dimensional embedded node feature, original denoising graph autoencoder, and deepNF. As for all these architectures, we train 200 epochs and use the extracted latent feature to train a SVM classifer with RBF kernel. We implement deepNF based on Pytorch and we confirmed deepNF hyperparameters setting with the author. Because we only run experiment in one protein interaction network, while the original deepNF integrate multiple protein interaction networks together. This explains why deepNF performance here is low. Another explanation may be from BioGRID network since to our knowledge, no paper is based on such dataset and then we don't have baseline result.

Figure 5.15 shows after incorporating embedded vector representation, we can see considerable improvement (compare NEGraphAE and GraphAE). Additionally, our model outperforms deepNF over all the metrics.

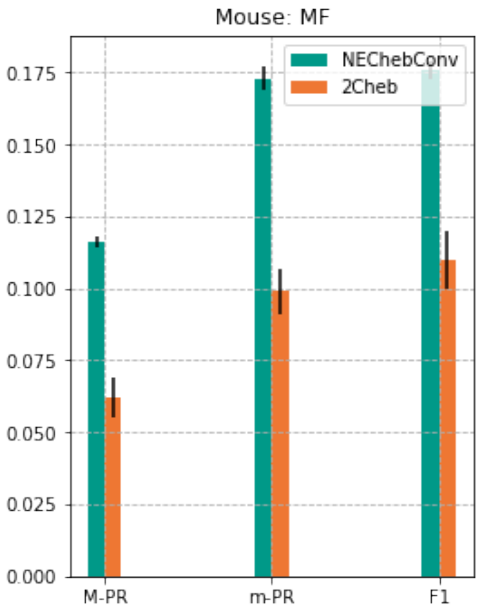


Figure 5.14: The performance of improved 2ChebyConv and original 2ChebyConv in analyzing mouse BioGRID networks with MF (117 annotations). Performance is measured by the area under the precision-recall curve,summarized over all GO terms both under the micro-averaging (m-AUPR), macro-averaging(M-AUPR) schemes and F1 score. The error bars are computed based on 5 trials

5.4.3 Conclusion

Because of the limitation of GPU resource, we can only provide experiments based on BioGRID mouse network because of using embedded feature introducing large node feature matrix. However, from experiments provided with regard to BioGRID mouse, it is enough to state our model outperforms the current baseline method, deepNF. We have already run some experiments on Cora citation network which we achieved comparable performance compared with current state-of-the-art model on this problem but they need to use additional hand-crafted feature. Since the underlying structure of cora citation network and protein interaction network are the same, we are more than convinced that our model would produce state-of-the-art results in protein function prediction this problem if GPU resource is enough.

For the next step, we would integrate batch node training into graph convolution networks, which then allows us to train large protein interaction networks.

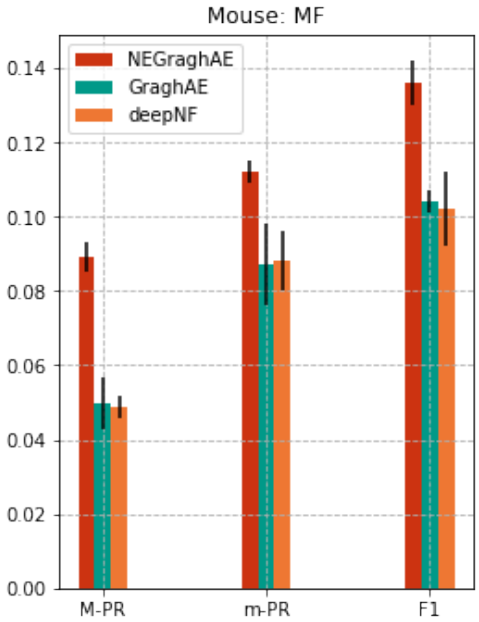


Figure 5.15: The performance of improved NEGraphAE, GraphAE and deepNF in analyzing mouse BioGRID networks with MF (117 annotations). Performance is measured by the area under the precision-recall curve,summarized over all GO terms both under the micro-averaging (m-AUPR), macro-averaging(M-AUPR) schemes and F1 score. The error bars are computed based on 5 trials.

Chapter 6

3D Facial Expression Recognition

Over the past a few years, most of works in face recognition involve only 2D images, but the recognition performance is typically largely affected by inherent pose and illumination variations. In order to deal with these issues, three-dimensional geometric of the human face is increasingly used because this avoids such pitfalls of 2D face recognition algorithms as change in lighting, different facial expressions, make-up and head orientation.

In this chapter, we seek to address the problem 3D facial expression recognition with geometric deep learning techniques Bronstein et al. [6] based on a new dataset, 4DFAB (2018 CVPR Cheng et al. [10]). We start by providing a strict problem definition, showing clearly the tasks, dataset and how do we structure different attributes. Then we show the approach we used to deal with this problem, from which we achieved the state-of-the-art performance compared to the baseline result provided by Cheng et al. [10]. Finally, we show details of how we design experiments to clearly indicate how we make progress step by step until finally achieved a good result. To our best knowledge, we are the first to build an end-to-end 3D facial expression recognition system.

6.1 Problem Definition

Given 3D mesh-structured training dataset with each mesh representing one of six facial expressions for one person, the problem is to recognize new 3D facial expressions from person not shown in the trainset. We use the 4DFAB dataset (Session1 [1]), containing 170 participants and each participant providing 4 to 6 basic facial expressions (*i.e.* anger, disgust, fear, happiness, sadness and surprise)(see Figure 1.5), resulting in a total of 1018 distinct expressions. Within these expressions, there are 5 *similar patterns* [2] for 1002 expressions, 4 similar patterns for 14 expressions, 3 similar patterns for 2 expressions. Therefore, the 4DFAB dataset contains 5072 patterns (meshes) in total. Data is partitioned in 10-folds, and 17 distinct participants in testset are not shown in trainset (with 153 distinct participants). The number of each class is balanced distributed in both trainset and testset.

[1] Session: There are 4 sessions in 4DFAB dataset, recording different video stimulus, and the same participant are invited to attend 4 times.

[2] Similar Patterns: The Frobenius norm of any two similar patterns is in [23.8, 36.9], mean is 32.24, while the Frobenius norm of two distinct expressions is in [241.2, 616.7], mean is 395.5.

Original Data The original meshes in 4DFAB are formulated as `menpo.TexturedTriMesh` [3] [4]. The main attributes we used include:

- `point`. 3D Cartesian coordinates with shape $[2064, 3]$, where 2064 means 2064 nodes in total for one mesh.
- `edge_indices`. An unordered index into points that rebuilds the edges of the mesh. There will be two edges present in cases where two triangles share an edge. The array shape is $[11970, 2]$.
- `label`. One label for each mesh in the range $[1, 6]$.

In order to take advantages of Pytorch, a dynamical deep learning framework with strong GPU acceleration, we encapsulate the above attributes into a Pytorch friendly class, named `Data`. We provide graph-based operations based on this class `Data`.

Data Class `Data` is constructed with the following attributes from `menpo.TexturedTriMesh`:

- `data.x (torch.FloatTensor)`. Save node feature matrix with shape [N, C], where N and C denotes number of nodes and feature dimension, respectively.
- `data.y (torch.LongTensor)`. Save the label of each data (mesh) in the range [0, 5] which is required for using cross entropy loss provided by Pytorch.
- `edge_index (torch.LongTensor)`. Save the graph connectivity into node index pairs with shape [2, 2*edges]. The indices in the first row are sorted from low to high.
- `data.pos (torch.FloatTensor)`. Save the 3-dimensional Cartesian coordinates, which is loaded directly from the property `point` of `menpo.TexturedTriMesh`.

6.2 Data Clean

From analysis of prediction results, we noticed the predicted labels for some data with similar patterns were not coincident, which were assumed to have same predicted labels with a well-trained network. After visualizing such kinds of data with Mayavi [5], we found they are corrupted (see Figure 6.1). By going through the whole dataset, the amount of such data is calculated to account for 5.4% of the whole dataset. It may influence our model's capacity to learn from data and inference. Therefore, we exclude this part of data, resulting in 4,799 meshes left in total. Without explicit statement, all the experiments in this chapter are conducted based on cleaned dataset except for one pair of experiments, which used to compare the performance between the cleaned and original dataset.

[3] Menpo: Menpo is a Python package designed from the ground up to make importing, manipulating and visualizing image and mesh data as simple as possible.

[4] menpo.TexturedTriMesh: https://menpo.readthedocs.io/en/stable/api/shape/TexturedTriMesh.html

[5] Mayavi: a scientific data visualizer written in Python, which supports for plotting 3D data.

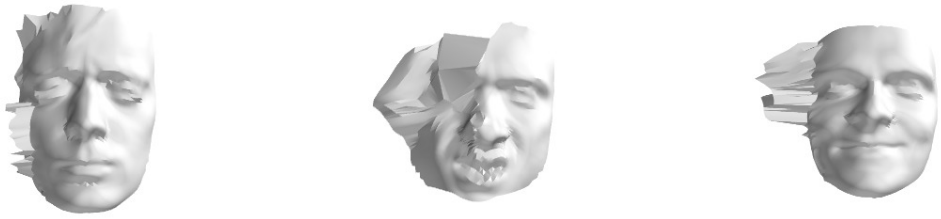


Figure 6.1: The front view of corrupted data.

6.3 Approach

In this section, we show our approach in detail and also provide the architecture (see Figure 6.2) which achieve the current state-of-the-art performance on this problem from our numerous experiments. We also report our whole exploration process shown in Section 6.5 step by step. The methods and implementation of basic modules with regard to the architecture figures shown in Sec. 6.5 are the same with what we are going to introduce in this section. Now we introduce our model in detail.

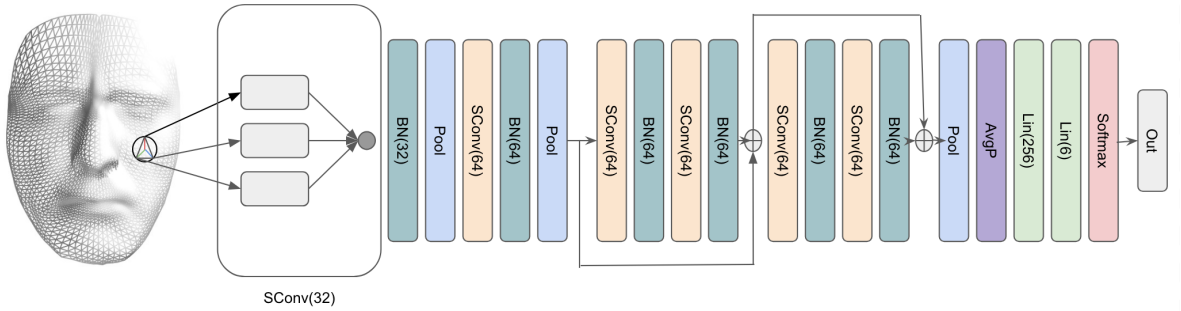


Figure 6.2: The residual mesh convolution architecture. PReLU activation function is applied on the output of each batch normalization layer expect for the last layer, where we apply softmax. It should be noting that non-linear activation function is applied after residual aggregation. Lin(o) represents 1×1 convolution solely operated on each node.

6.3.1 Notations and Definitions

In this section, We are interested in analyzing signals defined on an undirected, connected mesh, or say graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{U})$, which consists of a finite set of vertices $\mathcal{V}$ with $|\mathcal{V}| = n$, a set of edges $\mathcal{E}$, and $\mathcal{U} \in \mathbb{R}^{N \times N \times d}$ containing d-dimensional pseudo-coordinates $\mathbf{u}(i, j) \in \mathbb{R}^d$ where $j \in \mathcal{N}(i)$. $\mathcal{N}(i)$ denotes the neighborhood set of node i.

We denote $\mathbf{F} \in \mathbb{R}^{N \times d}$ as node feature matrix where each node containing d-dimensional

features. Then $\mathbf{f}_l$ is defined as the l^{th} channel feature map of node feature matrix $\mathbf{F}$ and $\mathbf{f}_l(i)$ represents the l^{th} channel feature of node i in $\mathcal{V}$.

6.3.2 Preprocessing

Since our model is based on BSplineConv (Fey et al. [18]), an intrinsic spatial convolution operator, we need firstly to scan the whole mesh to obtain pseudo-coordinates $\mathbf{u}(i,j)$, which is used to determine **how the features are aggregated**. Since data is already constructed as triangulated mesh (on the left of Figure 6.2), for any node i, we can compute all the corresponding pseudo-coordinates $\mathbf{u}(i,j)$ from traversing all the connected nodes j.

As for the problem of discrete manifolds, Fey et al. [18] suggest to use 3D Cartesian coordinates of the target point in respect to the origin point for each edge. We found it is better to use a globally normalized Cartesian coordinates as below:

$$\mathbf{u}(i,j) = 0.5 + \frac{pos_j - pos_i}{2 \cdot \max_{(v,w)\in\mathcal{E}} |pos_w - pos_v|} \tag{6.1}$$

Eq. 6.1 allows to map spatial relation to a fixed region [0, 1].

6.3.3 Weight Initialization

Weight Matrix We adopt the method introduced in He et al. [24] to initialize weight matrix in terms of SplineConv, Linear and batch normalization (here replace l_{in} with l_{out}) layer. Assume the input feature dimension in the k^{th} layer is l_{in}, the weight is then initialize with $\mathcal{N}(0, \mathrm{std})$, where std is defined as:

$$\mathrm{std} = \sqrt{\frac{2}{1 \times l_{in}}} \tag{6.2}$$

Bias We trivially initialize all the bias with constant value 0.

6.3.4 SplineConv

We use the following formula to aggregate feature into node i from neighbor nodes $j \in \mathcal{N}(i)$ (here we consider $\mathbf{N}(i)$ including i, *i.e.* aggregating both neighbor information and the central node feature):

$$\mathbf{g}_{l'}(i) = \frac{1}{|\mathcal{N}(i)|} \sum_{l=1}^{C^{k_1}} \left(\sum_{j\in\mathcal{N}(i)} \mathbf{f}_l(i) \sum_{\mathbf{p}\in\mathcal{P}(\mathbf{u}(i,j))} \mathcal{W}_{\mathbf{p},l,l'} \cdot \prod_{k=1}^{d} N_{k,p_i}^{m}(u_k) + b_{l,l'} \right) \quad l' = 1, \ldots, C^{k_2} \tag{6.3}$$

where N_{k,p_i}^{m} and $\mathcal{W}_{\mathbf{p},l,l'}$ denotes B-spline basis over degree m and learnable parameters, respectively.

6.3.5 Batch Normalization

We assume the output in k_2 layer is $\mathbf{G} \in \mathbb{R}^{N \times C^{k_2}}$. We take the idea of batch normalization [29], but now we would normalize the whole node feature matrix over each channel, which proved to be able to accelerate deep network training by reducing internal covariate shift.

$$\mathbf{g_l'} = \frac{\mathbf{g}_l - \mathbf{E}(\mathbf{g}_l)}{\sqrt{\mathbf{Var}(\mathbf{g}_l) + \epsilon}} \cdot \gamma + \beta \tag{6.4}$$

where $\mathbf{g}_l \in \mathbb{R}^{N \times 1}, \mathbf{E}(\mathbf{g}_l), \mathbf{Var}(\mathbf{g}_l)$ denotes the l^{th} column of the matrix $\mathbf{G}$, mean and variance of $\mathbf{g}_l$, respectively. γ and β represent learnable parameters. During training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation.

6.3.6 Graph Coarsening

Graph coarsening is what we denote as pool in the Figure 6.2. We use the graph coarsening strategy introduced in Section 3.3. We briefly explain how it works here.

The coarsening works as follows: given a graph, start with all nodes unmarked. Visit each vertex in a random order. For each vertex x, if x is not marked, merge x with the unmarked vertex y that maximizes (3.23) among all edges between x and unmarked vertices. Then mark x and y. If all neighbors of x have been marked, mark x and do not merge it with any vertex. Once all vertices are marked, the coarsening for this level is compete.

It should be noting that **this pooling strategy is unstable because of the greedy strategy of coarsening**, which means the same topology of different input mesh may yield different topology of output data after pooling. This leads to a different architecture compared with classification problems in Euclidean domain, where pooling or strided convolution produce **fixed size of images** which allows it to vectorize 2D image and use fully connected layer to aggregate all the input neurons to predifined number of output neurons. However, now we cannot do in this way but using average pooling layer to aggregate information in different nodes to a vector. Our experiments show that if fully connected layer can be conducted in the same manner as Euclidean domain, the classification performance could be higher.

$$\max \ W_{x,y} \left(\frac{1}{d_x} + \frac{1}{d_y} \right) \tag{6.5}$$

Where the edge weight $W_{x,y}$ is defined as l_2 distance.

6.3.7 Average Pooling

This layer average node feature matrix $\mathbf{F} \in \mathbb{R}^{N \times d}$ into a vector representation from trivially averaging. We assume the output vector is $\mathbf{g} \in \mathbb{R}^{1 \times d}$:

$$\mathbf{g} = \sum_{k=1}^{N} \mathbf{F}_{k^{\text{th}}\text{row}} \tag{6.6}$$

6.4 Training

In this section, we discuss how we train the model.

6.4.1 Loss Function

Generally, the loss function for a classification task should be cross entropy loss. The only thing need to be considered is whether the weight is need to be specified. However, it only occurs when label is quite sparse. Therefore, in this problem, we use the standard cross entropy loss with initializing weight of each element as 1. The loss can be described as:

$$\mathcal{L}(\mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{i=1}^{N} \ell_i(x, y) \tag{6.7}$$

$$\ell_i(x, y) = -\log(\frac{\exp(x[y])}{\sum_j \exp(x[j])}) \tag{6.8}$$

$\mathbf{Y}$ denotes label matrix.

6.4.2 Optimizer

Our experiments (Sec. 6.5) show Adam optimizer at the initial training stage would accelarate model convergence, while switching Adam to SGD with momentum in the later stages would further improve model performance.

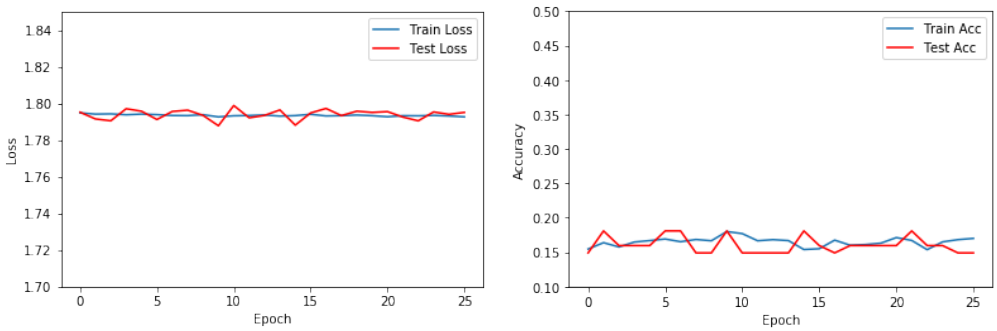
6.4.3 Adaptive Learning Rate Strategy

We found an adaptive learning rate strategy would not only lead to better classification, but also make the stage of hyperparameter tunning easier since it is not necessary to set annoying initial learning rate and defined learning rate decay steps, *e.g.* every 30 or 50 epochs etc.

For all our experiments, we train models with a decay learning rate and when test loss is in plateaus after k epochs, the learning rate r would decay by some factor α. Generally, we initialize learning rate at $r = 0.01$, patience $k = 10$ and factor $\alpha = 0.1$. This proves to take effectively for different models and task especially for Adam optimizer.

6.5 Experiments

In this section, we show all the key experiments that allow our explorations over this problem make progress.



(a) Loss (b) Accuracy

Figure 6.3: Experiment I with similar architecture and experiment setting to the paper of SplineCNN. 6.13(a) shows the comparison between train loss and test loss in 25 epoch; 6.13(c) shows the comparison between train accuracy and test accuracy in 25 epoch.

6.5.1 Experiment I

The similar experimental setup for shape correspondence of Fey et al. [18] is taken here. Since SplineConv proves to be advantageous for dealing with discrete manifolds problems, we expect it work in this problem. While shape correspondence is to make prediction for each node w.r.t 6890 classes, this problem requires to classify the whole mesh-structured data. Therefore, the node feature matrix of the output of the last convolutional Layer with the shape $N \times F$ needs to be either **mean aggregated** into $1 \times F$ or **reshape to a vector** with shape $1 \times (N \times F)$. The vector is then sent to FC layer to output with 6 classes. The first measure is adopted here, and the explanation is provided in experiment2.

Hence, the architecture with 6 convolutional layers: $\mathbf{SConv}((k1, k2, k3), 1, 32) \rightarrow \mathbf{SConv}((k1, k2, k3), 4 \times \mathbf{SConv}((k1, k2, k3), 64, 64) \rightarrow \mathbf{AvgP} \rightarrow \mathbf{FC}(256) \rightarrow \mathbf{FC}(6)$, where $\mathbf{AvgP}$ denotes a layer that averages features in the node dimension. The remaining setting is the same with Fey et al. [18]. As non-linear activation function, the *Exponential Linear Unit (ELU)* is used after each $\mathbf{SConv}$ and the first $\mathbf{FC}$ layer. For Cartesian coordinates the kernel size is chosen to be $k_1 = k_2 = k_3 = 4 + m = 5$. We only evaluate the case when degree $m = 1$. Training is done for 100 epochs with a batch size of 1, initial learning rate 0.01 and dropout probability 0.5, using the Adam optimizer [32] and cross entropy loss. The experiment is performed with NVIDIA GTX 1080.

Discussion Loss and accuracy w.r.t train and test shown in Figure 6.3. Training and test time for one epoch are 713.87s and 11.13s respectively. We found that after training with nearly 5 hours, the model still learns nothing useful to classify expressions and the test accuracy remained $1/6$. Analysis with the confusion matrix A.2, all expressions are recognized as the same expression (varied in different epochs).

Strategy We consider the most useful strategy to resolve the current problem should be using **larger batch size**, and the model complexity should be reduced to accelerate model training.

Analysis of mini-batch size Basically, with smaller mini-batch size, the higher variance or noise of gradient updates (by averaging the gradients in the mini-batch) is introduced, which proved to be helpful for converging to **flat minimizers** (compared to **shape minimizer** (see Figure 6.4)) with better generalization performance [31]. This does not mean the smaller (batch size is 1 in this experiment) the better since each gradient updated direction could be highly varied (Figure 6.5), which makes the training harder especially for dataset with lower variance (4DFAB compared to FAUST).

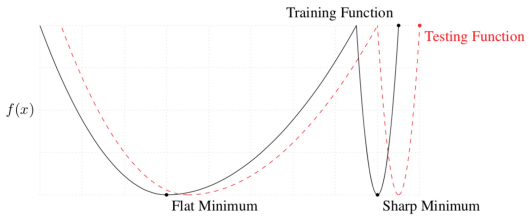


Figure 6.4: A Conceptual Sketch of Flat and Sharp Minima. The image is from Keskar et al. [31]

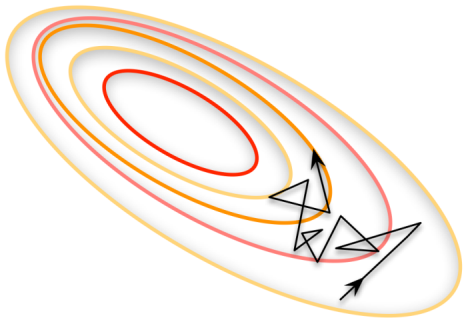


Figure 6.5: For non-linear deep neural networks, the local region of the cross section of the loss function is nearly a paraboloid. With batch size = 1, it tends to be hard for training since the momentum in each iteration is highly likely to be cancelled out. The view of gradient descent with batch size equal to 1.

6.5.2 Experiment II

With the discussion in the experiment I, we modified the experimental setup and additionally, set a pair of comparison experiments with *3 Convs without pooling* vs *3 Convs with pooling* (Figure 6.6), during which *SGD* and *Adam* are interleaving adopted so as to view the performance of these two commonly used optimizers.

For the experiment settings, ELU is used each convolution and pooling operation is the same with the section 1. Batch size is 16. The initial learning rate is 1e-2, divided by 10 when test loss is plateaus (no better loss found in predefined number of epochs). The initial optimizer is SGD with momentum = 0.9, and in the second stage, optimizer is changed to Adam. The two experiments are paralleled running on two NVIDIA GTX 1080.

Discussion Figure 6.7 and 6.8 show loss and accuracy for these two architectures. The training time each epoch for architecture 2.1 and 2.2 are **145.7s** and **93.2s** respectively. Table 6.1 shows the prediction accuracy for each class, and the confusion matrix (appendice A.3, A.4) provides detailed prediction statistics of each class. It's clear that:

1. **Optimizer**. Both models are hard to train with SGD at the initial stage with loss and accuracy remaining constant; after changed to Adam, train and test loss decrease

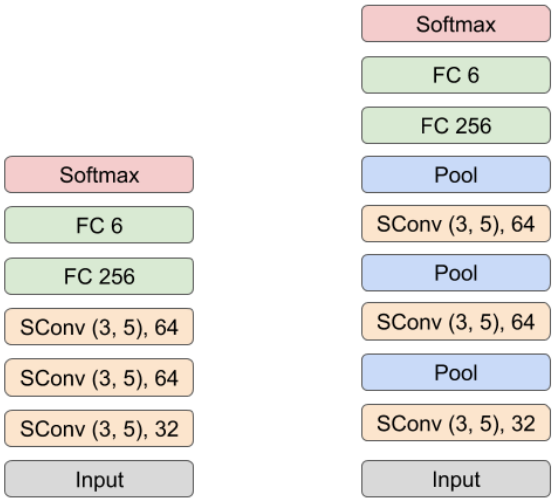


Figure 6.6: The architectures of 3conv layer with no pooling (architecture 2.1) and pooling (architecture 2.2) respectively from left to right. SConv (3, 5) means kernel dimension 3, kernel size 5 (B-spline basis degree of 1). The remaining experiments hold the same definition.

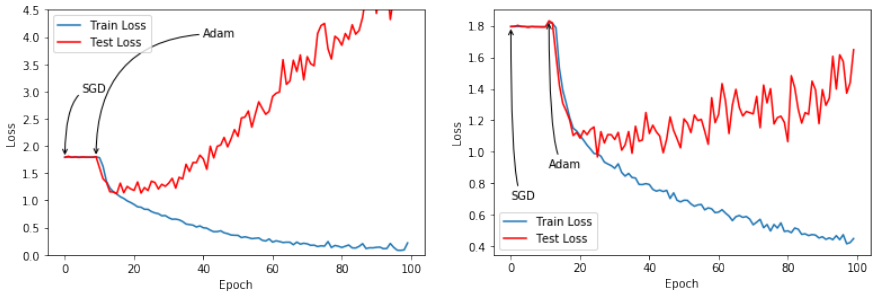
synchronously. It's worth noting that moving back to SGD for a well-trained model with Adam optimizer can normally produce better performance, though related experiment results not provided here.

2. **Overfitting**. While both architectures appear to overfitting after around 30 epochs, **pooling** structure helps avoid overfitting to some extent, and reduce training time remarkably (**93.2s** compared to **145.7s**). As for prediction accuracy, model with pooling structure is **4.6%** higher than that without pooling, and the mean accuracy after 50 epochs is around 62.4%, compared to 58.2%.

3. **Prediction performance for each class**. Table 6.1 shows the prediction accuracy of both architectures are uniform with respect to 6 classes, *i.e.* the prediction accuracy is highest for 'Happiness ' and 'Surprise', but low for 'Fear' and 'Sadness'. It is interpretable since expressions of some participants included in testset may be confusing or hard to distinguish.

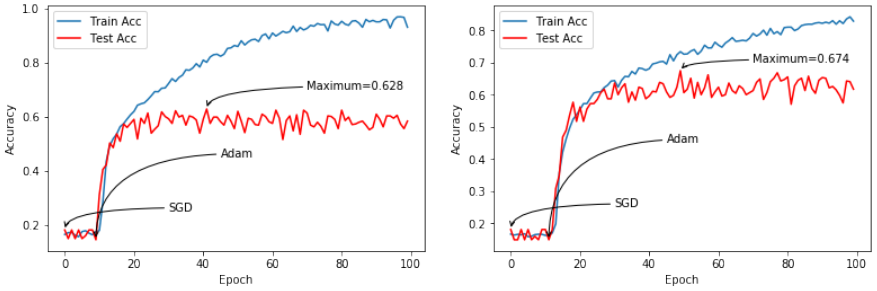
Analysis

- **Decay learning rate when test loss is in plateaus**. From experiments, we noticed that such an *adaptive learning rate* methods demonstrate better performance than *fixed learning rate* or *pre-scheduled learning rate*, and it requires much less effort in hyperparameter settings. With advisable fixed learning rate, the loss is able to converge at the beginning until trapped into a flat plateaus, which then requiring to reduce learning rate to some extent. However, it's quite annoying to find a good



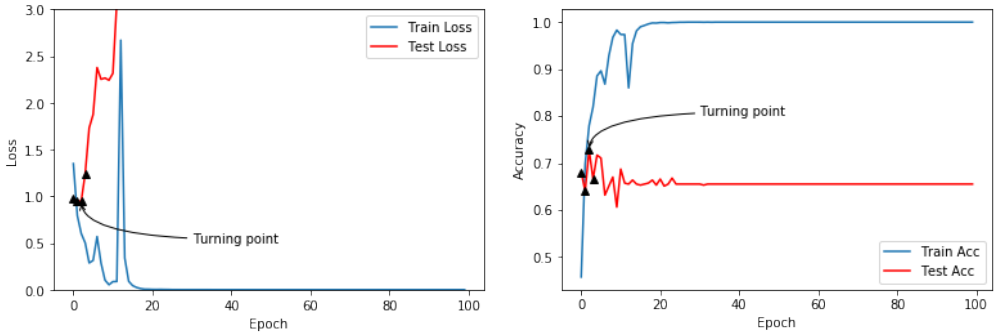
(a) 3Convs without pooling (b) 3Convs with pooling

Figure 6.7: The train and test loss for experiment II. Optimizer is changed to Adam from 11th epoch for both architectures.



(a) 3Convs without pooling (b) 3Convs with pooling

Figure 6.8: The train and test accuracy for experiment II. Optimizer is changed to Adam from 11th epoch for both architectures.



(a) Loss of 3convs with directly FC (b) Accuracy of 3convs with directly FC

Figure 6.9: The loss and accuracy of experiment 2* with directly FC structure.

	without pooling	with pooling
Anger	**60.00%**	56.00%
Disgust	**67.06%**	60.00%
Fear	40.00%	**61.43%**
Happiness	80.00%	**87.14%**
Sadness	44.71%	**64.71%**
Surprise	**83.53%**	76.47%
Total	62.8%	**67.5%**

Table 6.1: The prediction accuracy of experiment II for each class

	Parameters	GPU Memory (MB)
Directly FC	$792,582$	3.02
2FC After AvgP	$18,182$	0.07

Table 6.2: The comparison of parameters and GPU memory for two structures, namely directly FC and 2FC after AvgP. Parameters counted here are only from layers behind SConv. The memory for each parameter is 4bytes since it is saved as `torch.cuda.FloatTensor`.

initial learning rate and timing for reducing learning rate. Generally, we used initial learning rate with $1e-2$ and patience = 10, and this setting performs well for all experiments.

- **AvgP or Fully Connected Layer (FC)** (see Figure 6.10). When dealing with 2D image recognition, FC is generally added to the last Conv Layer to convert feature matrix with shape $W \times H \times D$ to fixed number of neurons so as to output C classes in the end. It's worth discussing whether or not this strategy is workable in Non-Euclidean domain since *node order is not defined in non-Euclidean domain*. Fey et al. [18] suggest AvgP and not using FC because methods should be *permutation-invariant*. From my opinion, FC actually is permutation-invariant, which parameterizes each entry of node feature matrix in a same way ignoring the location or spatial relation. The reason for not using FC here is mostly because the numbers of nodes of different data after 3 pooling layers are different because l_2 norms (weight) for each pair of nodes with same index in different data are different, affecting graclus coarsening which using a greedy strategy w.r.t weight and node degree (same for different data).

Experiment II*

In order to evaluate the performance of *directly applying FC without AvgP*, an additional comparison experiment made, with architecture: $\mathbf{SConv}((k1,k2,k3),1,32) \rightarrow \mathbf{SConv}((k1,k2,k3),32,64$ $\times\mathbf{SConv}((k1,k2,k3),64,64) \rightarrow \mathbf{FC}(6)$. The reason for removing the last 2 FC layers (com-

	without pooling	with pooling	directly FC
Anger	60.00%	56.00%	**69.33%**
Disgust	**67.06%**	60.00%	62.35%
Fear	40.00%	**61.43%**	58.57%
Happiness	80.00%	87.14%	**91.43%**
Sadness	44.71%	64.71%	**75.29%**
Surprise	**83.53%**	76.47%	80.00%
Total	62.8%	67.5%	**72.8%**

Table 6.3: The prediction accuracy for each class of three architectures

	without pooling	with pooling	directly FC
Training Time (1 epoch)	145.7s	93.2s	**226.8s**

Table 6.4: The training time of 1 epoch for each architecture. All experiments are performed on the same NVIDIA GTX 1080.

pared to architecture 2.1 and 2.2) is to keep the same order of magnitude of parameters so that the comparison is fair (Table 6.2). Figure 6.9 shows the rate of convergence is *extremely fast*. After 1 epoch, the prediction accuracy achieves **68.1%** and loss is reduced to **0.973**; at epoch 3, the accuracy reaches the highest with **72.8%** and then lowest loss **0.950** The downside is that this structure is much more prone to overfitting. After epoch 3, test loss starts to increase and train loss keep decreasing until reaching 0. Table 6.3 and 6.4 show the prediction accuracy for each class and training time of three architectures respectively. Table (A.5) provides detail prediction information for each class with confusion matrix. It's obviously that directly FC structure is capable of **improving model learning performance** and *generalization performance*, while pooling structure can *alleviate overfitting* to some extent, *stabilize the model* and *reduce training time*.

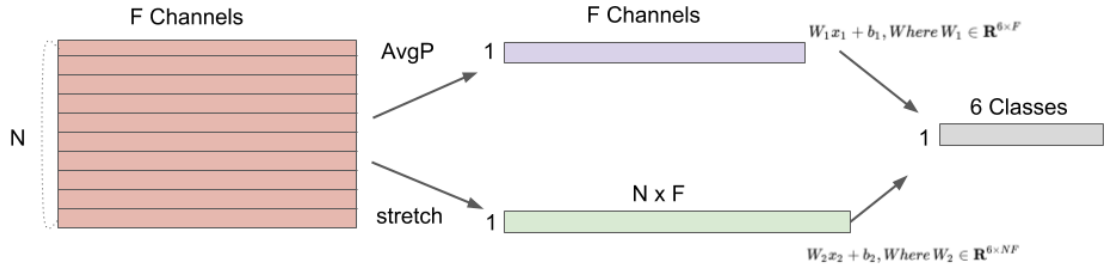


Figure 6.10: The view of transforming node feature matrix to vector with AvgP or directed stretch and then final fully connected layer. After pooling, N generally is varied, leading to a variable length vector $N \times F$, which makes it hard to initialize a FC layer.

Strategy The directions worthy to explore is to add regularization (data augmentation, batch normalization, dropout) and more layers to resolve the problem of overfitting and improve model performance with the same convolutional operation. It's beneficial to keep pooling structure, which reduces training time remarkably and raises prediction accuracy, but it also means a directly FC is not allowed without adding AvgP. The issue left is how to train model effectively with more layers since experiment 1 shows it's hard to capture underlying features with deeper architecture.

6.5.3 Experiment III

We use 6 convolutional layers (compared to 3 convs in experiment 2) in this part (Figure 6.11), and batch normalization, data augmentation, L2 weight decay are adopted as regularization strategy. Inspiration of additionally identity mapping in architecture 3.2 and 3.3 comes from ResNet [25], which stems from the idea of deeper model should produce no higher training error than its shallower counterpart. Such modification enables model to optimize easier and to gain accuracy. Identity mapping is defined as:

$$y = \mathcal{F}(x, W_i) + x \tag{6.9}$$

Where x and y are the input and output vectors of the layers. The function $\mathcal{F}(x, W_i)$ represents the residual mapping to be learned. In order to alleviate the negative influence of AvgP ($N \times F$ features are simply reduced to $1 \times F$), architecture 3.3 increased feature channels in order to keep more neurons after AvgP. A $1D$ convolution is performed (on node feature matrix) to match the input and output feature dimensions:

$$y = \mathcal{F}(x, W_i) + W_s x \tag{6.10}$$

Training was done using the SGD optimizer with momentum 0.9, initial learning rate 0.01, divided by 10 when in plateaus in 10 epochs, and L2 regularization 1e-5. Model is initialized with He et al. [24]. As before, cross entropy is used. Experiments were performed with GTX NVIDI 1080.

Discussion Figure 6.12 and 6.13 show loss and accuracy of three architectures. The prediction accuracy for three architectures are **68.5%**, **75.3%**, **71.5%** respectively. Currently, the state of the art result is provided by Cheng et al. [10] with **70.27%** for Session 1. The training times for each epoch are **148s**, **146.6s**, **557.7s** respectively. It should be noticed that:

1. **Regularizations and overfitting.** As we have already discussed in experiment 2, regularizations required to address the problem of overfitting, but it requires discussion about the capacity or performance of varies regularization strategies (dropout, batch normalization, data augmentation, weight decay), all of which can be simply considered as *adding random noise in the training process* and *marginalizing over the noise in testing process*. Some incomplete experiments are taken to tell the difference these strategies (not shown here). The results show that use *batch normalization* when removing Dropout, the effect is much *faster learning without a loss in generalization*.

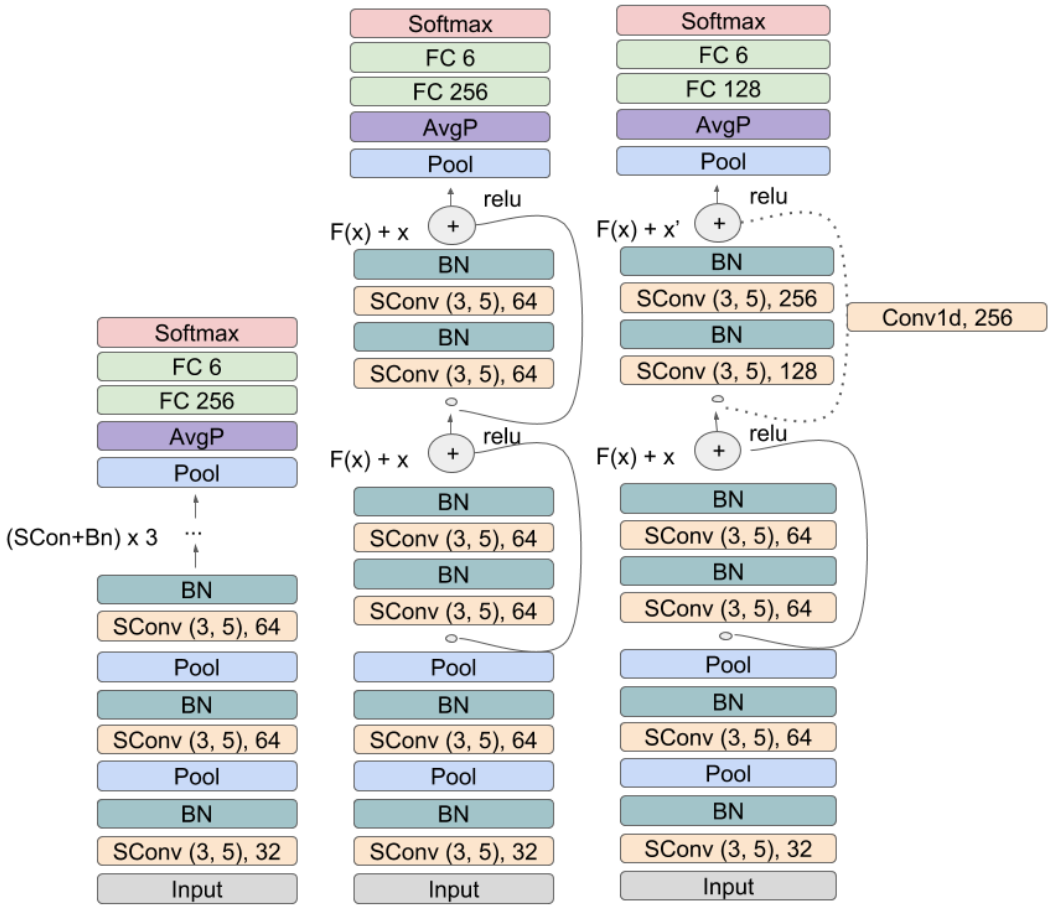
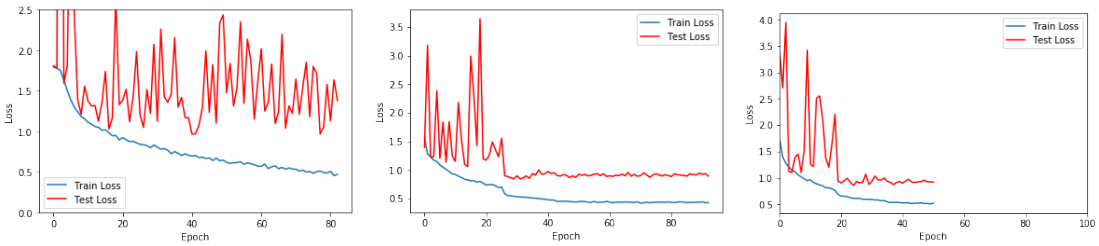


Figure 6.11: Architectures 3.1, 3.2, 3.3 respectively for experiment 3. ReLU activation function is used after every SConv.

This is particular remarkable when moving from 3layers to 6layers. Moreover, better backbone architecture looks more important than directly adding regularizations (See Figure 6.12), though all of three architectures behave better compared to experiment 2.

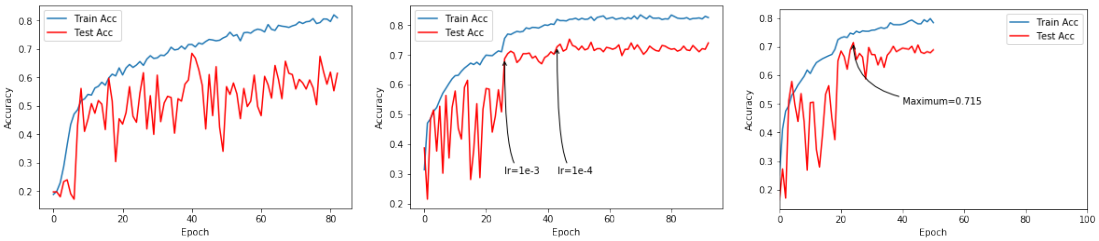
2. **Backbone architecture.** Figure 6.13 shows architectures with identity mapping are easier to optimize, and can gain accuracy from considerably increased depth. Because of time is limited, experiment of architecture 3.3 was terminated before 100 epochs. I expected better performance of architecture 3.3 compared to architecture 3.2, since more features are kept after averaging node features, but from the results, it isn't clear.

3. **Prediction accuracy for each class and confusion matrix.** All the experiments (Table 6.5) show that predictions for 'Surprise' and 'Happiness' tend to be more accuracy, and for 'Fear', 'Sadness' are still low. However, when compared with experiment 2, the improvement is significant. When analysis with confusion matrix Table (A.6, A.7, A.8), it is clearer that *'Sadness'* is mostly mistakenly recognized as *'Anger'*; *'Anger'* is



(a) Architecture 3.1 with batch normalization (b) Architecture 3.2 with identity mapping (c) Architecture 3.3 with dimension changing

Figure 6.12: The train and test loss for experiment 3.



(a) Architecture 3.1 with batch normalization (b) Architecture 3.2 with identity mapping (c) Architecture 3.3 with dimension changing

Figure 6.13: The train and test accuracy for experiment 3.

tended to be recognized as *'Disgust'* and *'Sadness'*; however, there is considerable number of *'Fear'* mistakenly recognized as *'Surprise'*, which is abnormal. It requires time to check the actual look and labels of original data.

4. **Performance comparison between cleaned and original data.** Figure 6.14 shows 4.9% prediction accuracy improvement after clean data. This proves the hypothesis that such erosion input data would affect the model learning.

6.5.4 Conclusion

In this section, we first introduce our proposed model which achieved state-of-the-art performance on 4DFAB dataset. Then we illustrate our experiments design in detail. From experiments, we found one issue in this field: *pooling* and *directly FC structure (without AvgP)* improve model performance and accelerate training process, but the new variant of pooling strategy for mesh-structured data with stable downsampling property is required so that the model can integrate pooling operation and directly FC altogether. We choose an alternative way from incorporating residual structure and better training strategies (*e.g.* initialization, batch normalization, combine Adam and SGD, adaptive learning rate decay, etc).

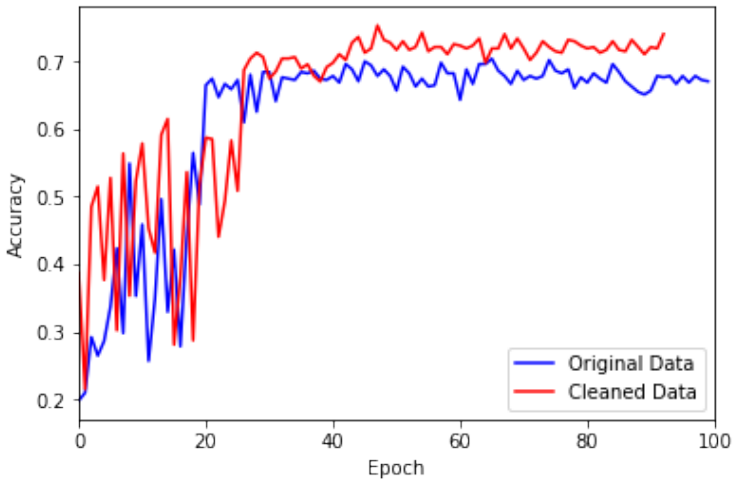


Figure 6.14: Train and test with original and cleaned data using architecture 3.2.

	Architecture 3.1	Architecture 3.2	Architecture 3.3
Anger	50.61%	**77.33**%	60.00%
Disgust	60.00%	**75.29**%	71.76%
Fear	48.57%	**61.43**%	52.86%
Happiness	85.71%	81.43%	**88.57**%
Sadness	**72.94**%	63.53%	62.35%
Surprise	90.59%	**91.76**%	**91.76**%
Total	68.5%	**75.3**%	71.5%

Table 6.5: The prediction accuracy of experiment 2 for each class

Chapter 7

Conclusion and Future Directions

The recent emergence of *geometric deep learning* allows us to bring the prosperity of deep learning to various communities and application domains. We have already seen these approaches have consistently push the state-of-the-art on tasks such as citation network prediction and shape correspondence. In this thesis, we use geometric deep learning to address two challenges arising in computational biology (*i.e.* protein function prediction) and computer graphics (*i.e.* 3D facial expression recognition) for decades. The underlying data structures for these two problems are graphs and manifolds, which constitute backbone of geometric deep learning. We expect the methods proposed in this thesis could benefit for applications in other domains.

In this thesis, we begin with providing *Euclidean deep learning* basis and the *graph theory*. We should stress that thoroughly understanding of Chapter 2 plays a vital role for understanding the *high-level design intuition* of the methods we review in Chapter 3. We introduce the state-of-the-art progress with regard to spectral and spatial graph convolution in Chapter 4. In this chapter, we also provide detailed discussion and analysis to compare the performance, difference of each operator. In particular, we find GCN, ChebyConv are superior than others when the underlying data structure is graph. On the other hand, spatial convolutions such B-splineConv are believed to be a better choice for dealing with problems on manifolds. The analysis and experimental conclusions in this Chapter assist us to design better methods when dealing with protein function prediction and 3D facial expression recognition.

In Chapter 5, we start from designing a multi-layer graph convolutional network to deal with the protein this problem, but the results are not as expecting. We then resort to build an denosiing graph aotuencoder so that we can take use of the power of SVM over the task of multi-label classification. We still don't get the results which we assumed. Finally, we find the problem and design a deep graph neural network, which is capable of learning graph high-dimensional topological structure and make prediction over nodes. We believe such model can also be applied to other fields or problems, where the underlying data is structured as graph.

In Chapter 6, we design a efficient residual graph convolutional network to recognize 3D facial expressions. To our best knowledge, we are the first to bring geometric deep learning in this domain. In this chapter, we provide experiments in detail, which show clearly how we make progress and how we find and solve problems step by step.

7.1 Future Directions

7.1.1 Bach training large-scale network with graph convolution

When we dealt with protein function prediction using our proposed deep graph neural networks, we were held back by high GPU memory requirement because we cannot train model with a batch size nodes at each iteration. For example, if we consider about chebyshev convolution, the laplancian matrix $\mathbf{L} \in \mathbb{R}^{N \times N}$ is required while it is not reasonable to train with only a small part of nodes. If we consider about spatial convolution such as BSplineConv, the aggregation process also relies on edge connectivity, while randomly selecting a batch nodes cannot guarantee it.

7.1.2 Differentiable and Stable Mesh Pooling Strategy

Since the general pooling choice of Graclus graph coarsening would normally lead to outputs with different topology, we then need to study about stable mesh pooling strategy. We have already proved that this would accelerate model training and improve performance.

On the other hand, differentiable pooling or subsampling and upsampling method are important, which is the key for designing 3D or high-dimensional generative models.

7.1.3 3D Generative Models

In the recent years, we have already seen the prosperity of generative models in 2D image domain, while it would produce more interesting and significant work if we make it possible to build 3D mesh generative models, or graph generative models. Currently, we are already engaged in this field and already achieved some good results with MeshVAE and MeshGANs.

Appendix A

Ethics Checklist

	Yes	No
Section 1: HUMAN EMBRYOS/FOETUSES		
Does your project involve Human Embryonic Stem Cells?		✓
Does your project involve the use of human embryos?		✓
Does your project involve the use of human foetal tissues / cells?		✓
Section 2: HUMANS		
Does your project involve human participants?		✓
Section 3: HUMAN CELLS / TISSUES		
Does your project involve human cells or tissues? (Other than from Human Embryos/Foetuses i.e. Section 1)?		✓
Section 4: PROTECTION OF PERSONAL DATA		
Does your project involve personal data collection and/or processing?		✓
Does it involve the collection and/or processing of sensitive personal data (e.g. health, sexual lifestyle, ethnicity, political opinion, religious or philosophical conviction)?		✓
Does it involve processing of genetic information?		✓
Does it involve tracking or observation of participants? It should be noted that this issue is not limited to surveillance or localization data. It also applies to Wan data such as IP address, MACs, cookies etc.		✓
Does your project involve further processing of previously collected personal data (secondary use)? For example Does your project involve merging existing data sets?		✓
Section 5: ANIMALS		
Does your project involve animals?		✓
Section 6: DEVELOPING COUNTRIES		
Does your project involve developing countries?		✓

If your project involves low and/or lower-middle income countries, are any benefit-sharing actions planned?		✓
Could the situation in the country put the individuals taking part in the project at risk?		✓
Section 7: ENVIRONMENTAL PROTECTION AND SAFETY		
Does your project involve the use of elements that may cause harm to the environment, animals or plants?		✓
Does your project deal with endangered fauna and/or flora /protected areas?		✓
Does your project involve the use of elements that may cause harm to humans, including project staff?		✓
Does your project involve other harmful materials or equipment, e.g. high-powered laser systems?		✓
Section 8: DUAL USE		
Does your project have the potential for military applications?		✓
Does your project have an exclusive civilian application focus?		✓
Will your project use or produce goods or information that will require export licenses in accordance with legislation on dual use items?		✓
Does your project affect current standards in military ethics e.g., global ban on weapons of mass destruction, issues of proportionality, discrimination of combatants and accountability in drone and autonomous robotics developments, incendiary or laser weapons?		✓
Section 9: MISUSE		
Does your project have the potential for malevolent/criminal/terrorist abuse?		✓
Does your project involve information on/or the use of biological-, chemical-, nuclear/radiological-security sensitive materials and explosives, and means of their delivery?		✓
Does your project involve the development of technologies or the creation of information that could have severe negative impacts on human rights standards (e.g. privacy, stigmatization, discrimination), if misapplied?		✓
Does your project have the potential for terrorist or criminal abuse e.g. infrastructural vulnerability studies, cybersecurity related project?		✓
Section 10: LEGAL ISSUES		
Will your project use or produce software for which there are copyright licensing implications?		✓

Will your project use or produce goods or information for which there are data protection, or other legal implications?		✓
Section 11: OTHER ETHICS ISSUES		
Are there any other ethics issues that should be taken into consideration?		✓

A.1 Brief Statement

I confirm all the ticks on the above *ethics checklist* are made by my own and are consistent with this project. We use the following dataset in this thesis, *i.e.* [50], 4, BioGRID (https://thebiogrid.org/) and STRING https://string-db.org/ protein interaction network, 4DFAB [10], all of which are publicly available.

List of Tables

		Predicted Class					
		Anger	Disgust	Fear	Happiness	Sadness	Surprise
Actual Class	Anger	0	0	0	75	0	0
	Disgust	0	0	0	85	0	0
	Fear	0	0	0	70	0	0
	Happiness	0	0	0	70	0	0
	Sadness	0	0	0	85	0	0
	Surprise	0	0	0	85	0	0

Table A.2: Confusion matrix of experiment 1 in epoch 25

		Predicted Class					
		Anger	Disgust	Fear	Happiness	Sadness	Surprise
Actual Class	Anger	45	20	0	1	5	4
	Disgust	0	57	17	7	1	3
	Fear	0	9	28	4	12	17
	Happiness	4	7	2	56	0	1
	Sadness	24	13	8	2	38	0
	Surprise	1	0	8	0	5	71

Table A.3: Confusion matrix of experiment 2 for 3Convs without pooling

		Predicted Class					
		Anger	Disgust	Fear	Happiness	Sadness	Surprise
Actual Class	Anger	42	12	3	1	16	1
	Disgust	0	51	23	1	6	4
	Fear	2	4	43	0	3	18
	Happiness	4	2	3	61	0	0
	Sadness	16	4	10	0	55	0
	Surprise	0	0	20	0	0	65

Table A.4: Confusion matrix of experiment 2 for 3Convs with pooling

		Predicted Class					
		Anger	Disgust	Fear	Happiness	Sadness	Surprise
Actual Class	Anger	52	12	0	0	11	0
	Disgust	5	53	14	5	8	0
	Fear	0	6	41	5	5	13
	Happiness	0	0	0	64	5	1
	Sadness	21	0	0	0	64	0
	Surprise	0	0	17	0	0	68

Table A.5: Confusion matrix of experiment 2 for 3Convs with directly FC

		Predicted Class					
		Anger	Disgust	Fear	Happiness	Sadness	Surprise
Actual Class	Anger	38	6	8	0	23	0
	Disgust	0	51	20	5	4	5
	Fear	0	1	34	5	2	28
	Happiness	7	1	2	60	0	0
	Sadness	16	0	5	1	62	1
	Surprise	0	0	4	0	4	77

Table A.6: Confusion matrix of experiment 2 for architecture 3.1

		Predicted Class					
		Anger	Disgust	Fear	Happiness	Sadness	Surprise
Actual Class	Anger	58	14	2	0	1	0
	Disgust	0	64	10	5	5	1
	Fear	0	3	43	2	2	20
	Happiness	2	6	1	57	4	0
	Sadness	24	2	3	0	54	2
	Surprise	0	0	2	0	5	78

Table A.7: Confusion matrix of experiment 2 for architecture 3.2

		Predicted Class					
		Anger	Disgust	Fear	Happiness	Sadness	Surprise
Actual Class	Anger	45	15	4	0	11	0
	Disgust	0	61	11	4	9	0
	Fear	2	6	37	2	3	20
	Happiness	4	1	0	62	3	0
	Sadness	29	3	0	0	53	0
	Surprise	0	0	4	0	3	78

Table A.8: Confusion matrix of experiment 2 for architecture 3.3

List of Figures

Bibliography

[1] Ambady, N. and Rosenthal, R. (1992). Thin slices of expressive behavior as predictors of interpersonal consequences: A meta-analysis. *Psychological bulletin*, 111(2):256. pages 6

[2] Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1. pages 1

[3] Bhagat, S., Cormode, G., and Muthukrishnan, S. (2011). Node classification in social networks. In *Social network data analytics*, pages 115–148. Springer. pages 1

[4] Bogo, F., Romero, J., Loper, M., and Black, M. J. (2014). Faust: Dataset and evaluation for 3d mesh registration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3794–3801. pages 41, 48, 92

[5] Boscaini, D., Masci, J., Rodolà, E., and Bronstein, M. (2016). Learning shape correspondence with anisotropic convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 3189–3197. pages 26, 36, 38

[6] Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. (2017). Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42. pages i, 3, 30, 48, 50, 72, 98

[7] Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2013). Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*. pages 26, 29, 30, 32

[8] Chan, P. K., Schlag, M. D., and Zien, J. Y. (1994). Spectral k-way ratio-cut partitioning and clustering. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 13(9):1088–1096. pages 24

[9] Chang, C.-C. and Lin, C.-J. (2011). Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27. pages 6, 64, 65

[10] Cheng, S., Kotsia, I., Pantic, M., and Zafeiriou, S. (2017). 4dfab: A large scale 4d facial expression database for biometric applications. *arXiv preprint arXiv:1712.01443*. pages i, iii, 6, 72, 84, 92

[11] Cho, H., Berger, B., and Peng, J. (2016). Compact integration of multi-network topology for functional analysis of genes. *Cell systems*, 3(6):540–548. pages 4, 55, 60, 65

[12] Chung, F. R. (1997). *Spectral graph theory*. Number 92. American Mathematical Soc. pages 16

[13] Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*. pages 12

[14] Coifman, R. R. and Maggioni, M. (2006). Diffusion wavelets. *Applied and Computational Harmonic Analysis*, 21(1):53–94. pages 18

[15] Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852. pages 26, 27, 28, 29, 33, 34, 44, 51, 98

[16] Dhillon, I. S., Guan, Y., and Kulis, B. (2007). Weighted graph cuts without eigenvectors a multilevel approach. *IEEE transactions on pattern analysis and machine intelligence*, 29(11). pages 24, 29

[17] Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pages 2224–2232. pages 3

[18] Fey, M., Lenssen, J. E., Weichert, F., and Müller, H. (2017). Splinecnn: Fast geometric deep learning with continuous b-spline kernels. *arXiv preprint arXiv:1711.08920*. pages 26, 36, 40, 41, 42, 51, 75, 78, 82, 98

[19] Gligorijević, V., Barot, M., and Bonneau, R. (2017). deepnf: Deep network fusion for protein function prediction. *bioRxiv*, page 223339. pages 5, 55, 60, 61, 65

[20] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. pages 61

[21] Grady, L. J. and Polimeni, J. R. (2010). *Discrete calculus: Applied analysis on graphs for computational science*. Springer Science & Business Media. pages 15

[22] Graham, B. (2014). Fractional max-pooling. *arXiv preprint arXiv:1412.6071*. pages 11

[23] Hammond, D. K., Vandergheynst, P., and Gribonval, R. (2011). Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150. pages 21, 22, 23, 33

[24] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034. pages 11, 61, 64, 75, 84

[25] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778. pages 35, 84

[26] Henaff, M., Bruna, J., and LeCun, Y. (2015). Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*. pages 26, 31, 32

[27] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*. pages 52

[28] Horn, R. A., Horn, R. A., and Johnson, C. R. (1990). *Matrix analysis*. Cambridge university press. pages 20

[29] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*. pages 63, 76

[30] Karypis, G. and Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392. pages 24

[31] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*. pages 79, 101

[32] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. pages 29, 44, 78

[33] Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*. pages 13

[34] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*. pages 3, 26, 35, 44, 51, 97

[35] Kokkinos, I., Bronstein, M. M., Litman, R., and Bronstein, A. M. (2012). Intrinsic shape context descriptors for deformable shapes. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 159–166. IEEE. pages 43, 93

[36] Lafon, S. and Lee, A. B. (2006). Diffusion maps and coarse-graining: A unified framework for dimensionality reduction, graph partitioning, and data set parameterization. *IEEE transactions on pattern analysis and machine intelligence*, 28(9):1393–1403. pages 24

[37] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436. pages 2, 8, 97

[38] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. pages 2

[39] Liben-Nowell, D. and Kleinberg, J. (2007). The link-prediction problem for social networks. *journal of the Association for Information Science and Technology*, 58(7):1019–1031. pages 1

[40] Litman, R. and Bronstein, A. M. (2014). Learning spectral descriptors for deformable shape correspondence. *IEEE transactions on pattern analysis and machine intelligence*, 36(1):171–180. pages 43

[41] Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. pages 64

[42] Masci, J., Boscaini, D., Bronstein, M., and Vandergheynst, P. (2015). Geodesic convolutional neural networks on riemannian manifolds. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 37–45. pages 26, 36, 37, 48

[43] Monti, F., Boscaini, D., Masci, J., Rodola, E., Svoboda, J., and Bronstein, M. M. (2017a). Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proc. CVPR*, volume 1, page 3. pages 36, 39, 40, 44, 98

[44] Monti, F., Bronstein, M., and Bresson, X. (2017b). Geometric matrix completion with recurrent multi-graph neural networks. In *Advances in Neural Information Processing Systems*, pages 3700–3710. pages 3, 26

[45] Mostafavi, S. and Morris, Q. (2012). Combining many interaction networks to predict gene function and analyze gene lists. *Proteomics*, 12(10):1687–1696. pages 4, 60

[46] Mostafavi, S., Ray, D., Warde-Farley, D., Grouios, C., and Morris, Q. (2008). Genemania: a real-time multiple association network integration algorithm for predicting gene function. *Genome biology*, 9(1):S4. pages 4

[47] Narang, S. K. and Ortega, A. (2012). Perfect reconstruction two-channel wavelet filter banks for graph structured data. *IEEE Transactions on Signal Processing*, 60(6):2786–2799. pages 17

[48] Pantic, M., Nijholt, A., Pentland, A., and Huanag, T. S. (2008). Human-centred intelligent human? computer interaction (hci^2): how far are we from attaining it? *International Journal of Autonomous and Adaptive Communications Systems*, 1(2):168–187. pages 6

[49] Piegl, L. and Tiller, W. (2012). *The NURBS book*. Springer Science & Business Media. pages 40

[50] Rao, N., Yu, H.-F., Ravikumar, P. K., and Dhillon, I. S. (2015). Collaborative filtering with graph information: Consistency and scalable methods. In *Advances in neural information processing systems*, pages 2107–2115. pages 92

[51] Salti, S., Tombari, F., and Di Stefano, L. (2014). Shot: Unique signatures of histograms for surface and texture description. *Computer Vision and Image Understanding*, 125:251–264. pages 43

[52] Scherer, D., Müller, A., and Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pages 92–101. Springer. pages 11

[53] Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., and Eliassi-Rad, T. (2008). Collective classification in network data. *AI magazine*, 29(3):93. pages 41, 44, 50

[54] Shi, J. and Malik, J. (2000). Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905. pages 24

[55] Shuman, D. I., Ricaud, B., and Vandergheynst, P. (2012). A windowed graph fourier transform. In *Statistical Signal Processing Workshop (SSP), 2012 IEEE*, pages 133–136. Ieee. pages 22

[56] Smola, A. J. and Kondor, R. (2003). Kernels and regularization on graphs. In *Learning theory and kernel machines*, pages 144–158. Springer. pages 19

[57] Spielman, D. A. (2007). Spectral graph theory and its applications. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*, pages 29–38. IEEE. pages 20

[58] Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*. pages 11

[59] Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. (2015). Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee. pages 67

[60] Tang, S., Wang, X., Lv, X., Han, T. X., Keller, J., He, Z., Skubic, M., and Lao, S. (2012). Histogram of oriented normal vectors for object recognition with a depth sensor. In *Asian conference on computer vision*, pages 525–538. Springer. pages 6

[61] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(Dec):3371–3408. pages 14

[62] Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., and Borgwardt, K. M. (2010). Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242. pages 1

[63] Zhu, X. and Rabbat, M. (2012a). Approximating signals supported on graphs. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 3921–3924. IEEE. pages 18

[64] Zhu, X. and Rabbat, M. (2012b). Graph spectral compressed sensing for sensor networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 2865–2868. IEEE. pages 18