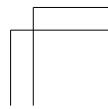
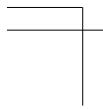
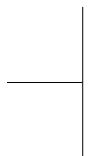
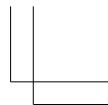
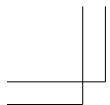


グラフ理論と Graph Neural Networks 概論

lib-arts 著

2020-03-18 版 発行



はじめに

初版刊行にあたって

グラフ理論 (Graph Theory) は様々な利用シーンで用いられています。例えば路線図や相関図のように、観測対象の位置よりも関係性を主に記述したい際にグラフは便利です。

3



図: 路線図 (東京メトロ)

<https://www.tokyometro.jp/station/>

上記は東京メトロのホームページから路線図を引っ張ってきました。地下鉄はそれぞれの駅の位置以上に路線によるつながりに着目した方がわかりやすいです。ここで、グラフ理論においては駅にあたるのがノード、路線にあ

3

たるのがエッジと呼んでいます。

このように、知っておくと非常に便利な表現であるグラフですが、近年このグラフにも DeepLearning を導入するという動きが進んでいます。特に画像を中心とする DeepLearningにおいて大きなブレークスルーを起こした CNN をベースにしたアルゴリズムのグラフ畳み込みネットワークが 2016 年頃から様々な研究が発表されるようになってきています。

グラフへの DeepLearning の応用 (Graph Neural Networks) に関しては様々な研究がなされているものの、なかなか体系的にまとめた文献が少ないということもあるので、本書ではグラフへの DeepLearning の導入に関して取りまとめていければと思います。参考資料としては、下記の Survey を元に取りまとめを行っていきます。

<https://arxiv.org/abs/1901.00596>

こちらは 2019 年の 1 月に出された Graph Neural Networks に関する Survey で、研究トピックを体系的にまとめられています。本書ではこちらの Survey の内容の解説や必要に応じて追記を行っていきます。

4

これを機に、Graph Neural Networks に関して学ぶきっかけとしていただけたら良いなど考えています。

本書の構成

本書の構成としては、まず第 1 章ではグラフ理論の基本的な解説を行ったのちに、Graph Neural Networks の大枠として、Graph Neural Networks に期待されることや、研究トピックの紹介を行っています。続く第 2 章では、Graph Neural Networks の手法について取り扱っています。具体的には Recurrent Graph Neural Networks(RecGNNs) と Convolutional Graph Neural Networks(ConvGNNs) の二つについて取り扱っています。

第 3 章では Graph Neural Networks の応用タスクとして、Graph AutoEncoders(GAEs) と Spatial Temporal Graph Neural Networks(STGNNs) についてご紹介します。第 4 章では、Graph Neural

Networks の実装として、NetworkX や PyTorch などをベースにした OSS のライブラリである、Deep Graph Library(DGL) の紹介と使い方、理論との対応などについて解説します。

全体を通して、グラフ理論の基礎から Graph Neural Networks の概論、仕組み、トレンド、実装がそれぞれ理解できるような構成としています。

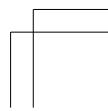
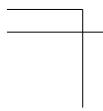
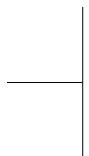
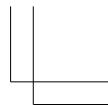
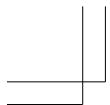
本書の対象読者

- ・CNNなどの基本知識がある方
- ・Pythonについては簡単に使用できる方 (Jupyter Notebook が使ったことがあれば大丈夫だと思います)
- ・Graph Neural Networksについて興味がある方

5

本書の記法について注意

・目次についている*は発展項目のため、初見の際は読み飛ばしても構わないです。余裕のある時にご確認いただくことで視野が広がるような話題をチョイスしています。



目次

はじめに	3
初版刊行にあたって	3
本書の構成	4
本書の対象読者	5
本書の記法について注意	5
第1章 序論 (Introduction)	9
1.1 グラフ理論の概要	9
1.1.1 グラフ理論の概要と歴史	9
1.1.2 隣接行列と有向グラフ、無向グラフ	11
1.2 応用例 (機械学習とグラフ理論)	12
1.2.1 グラフィカルモデル	12
1.2.2 TensorFlow と計算グラフ	14
1.3 Survey の概要と数式表記	15
1.3.1 Survey の概要と目次	15
1.3.2 グラフ理論の数式的な定義	17
1.4 Graph Neural Networks	20
1.4.1 背景 (Background)	21
1.4.2 GNN の研究トピックの紹介	22
1.5 第1章のまとめ	24

目次

第 2 章 Graph Neural Networks の手法	25
2.1 Recurrent GNNs	25
2.2 Convolutional GNNs	31
2.2.1 Spectral-based ConvGNNs(Section5-A)	33
2.2.2 Spatial-based ConvGNNs(Section5-B)	35
2.2.3 Graph Pooling Modules(Section5-C)	38
2.2.4 Discussion of Theoretical Aspects(Section5-D) . .	38
第 3 章 GNNs のタスク定義	39
3.1 GAEs(Graph AutoEncoders)	39
3.1.1 Network Embedding(Section6-A)	40
3.1.2 Graph Generation(Section6-B)	43
3.2 STGNNs	47
第 4 章 GNNs の実装 (Deep Graph Library)	53
4.1 インストール	53
4.2 動作確認	55
4.3 基本事項	62
4.3.1 Creating a graphについて	63
4.3.2 Assigning a featureについて	66
本書について	71
著者プロフィール	71
注意事項	71

第 1 章

序論 (Introduction)

9

第 1 章では、グラフ理論の基本的なトピックについて取り扱います。1-1 節でグラフ理論の概要、1-2 節でグラフ理論の応用例として機械学習文脈でのグラフ理論の適用、1-3 節で参考にする Survey の概要やグラフの数式定義についてまとめた上で、1-4 節では本書のテーマである、Graph Neural Networks の導入について簡単にご紹介します。また、グラフ理論にあまり馴染みがない方が一度思考の整理ができるように 1-5 節では第 1 章のまとめについて取りまとめました。

1.1 グラフ理論の概要

1.1.1 グラフ理論の概要と歴史

概要を抑えるにあたって、まずは Wikipedia の記載を確認します。

<https://ja.wikipedia.org/wiki/グラフ理論>

グラフ理論（グラフりょん、英: graph theory）は、ノード（頂点; Node）の集合とエッジ（辺; Edge）の集合で構成されるグラフに関する数学の理論である。グラフ（データ構造）などの応用がある。

9

上記は Wikipedia の冒頭を少々編集したものです。ここでキーワードとしてはノードとエッジです。

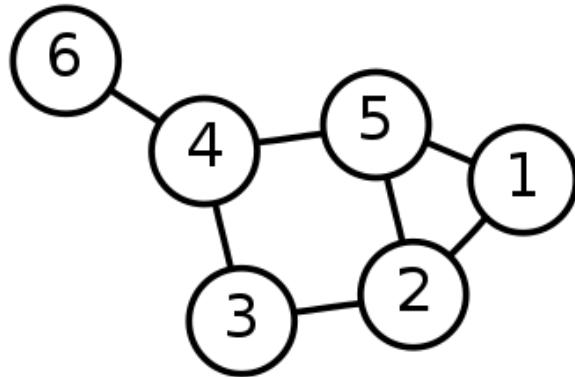


図: 1-1 グラフの図示

上図も Wikipedia の写真を拝借したのですが、この図は 6 つのノード (点) と 7 つのエッジ (辺) で成り立っています。

コンピュータでの取り扱いについては 1-1-2 節の隣接行列のところで説明しますが、行列表記との対応を覚えておくと便利です。また、ノードとエッジの組み合わせで表した上図をグラフと呼んでいます。この辺までは定義なので、そういうものだと覚えていただければと思います。

また、グラフ理論の歴史ですが、1736 年に「ケーニヒスベルクの問題」と呼ばれるパズルに対してオイラーが解法を示したのが起源のひとつとされるそうです。

<https://ja.wikipedia.org/wiki/一筆書き#ケーニヒスベルクの問題>

詳しくは上記を見ていただければと思いますが、18 世紀の初め頃にプロイセン王国の東部、東プロイセンの首都であるケーニヒスベルクという大きな町の橋を題材に一筆書きで橋が渡れるかという問題が題材となっています。始まりそのものが非常に具体的な話題なので、色々と直感が働かせやす

い分野なのではと思います。

以上が簡単な概要になります。続いて 1-1-2 節でグラフの行列的取り扱いの隣接行列について確認します。

1.1.2 隣接行列と有向グラフ、無向グラフ

隣接行列 (adjacency matrix) はグラフの行列表現になります。

<https://ja.wikipedia.org/wiki/隣接行列>

こちらも Wikipedia がわかりやすいので上記をベースに進めていきます。

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

図: 1-2 隣接行列 (Adjacency matrix)

言葉で説明するのが難しいので、結論から表現すると上記が 1-1-1 節で例に出したグラフの隣接行列です。こちらも Wikipedia に載っていたものをお借りしました。

さて、上記の意味ですが、行列の a 行 b 列が 1 の場合は a と b がエッジ

で連結しているということを意味しています。例えば 1 行 2 列が 1 なので 1 と 2 は連結している一方で、1 行 3 列が 0 なので 1 と 3 は連結していません（今回は 0 からではなく 1 から始まっていることに注意が必要です）。このように要素の数を行数と列数に持つ行列を用意することで、グラフを行列で表記することができます。これが隣接行列の基本的な考え方です。

また、グラフには向きがあるグラフと向きがないグラフがあります。それぞれ有向グラフ、無向グラフと言います。冒頭の例は向きがないので無向グラフ (Undirected Graph) です。例としてあげるなら駅の路線図などがわかりやすいかもしれません。有向グラフ (Directed Graph) は人物相関図など一方通行もありえる場合に用います。

それぞれの隣接行列については、無向グラフは a 行 b 列と b 行 a 列の値が同じなのに対し、有向グラフはそれは成り立ちません。このように a 行 b 列と b 行 a 列を区別することで、グラフ上の向きという概念を行列で表現します。

1.2 応用例 (機械学習とグラフ理論)

応用的な例をということで 1-2 節では機械学習との関連についてまとめられればと思います。1-2-1 節では PRML の 8 章を参考にグラフィカルモデル、1-2-2 節では TensorFlow の計算グラフと TensorBoard について取り扱います。

1.2.1 グラフィカルモデル

グラフィカルモデルに関する解説については PRML の 8 章が情報量が豊富で非常に良いです。

Figure 8.11 An extension of the model of Figure 8.10 to include Dirichlet priors over the parameters governing the discrete distributions.

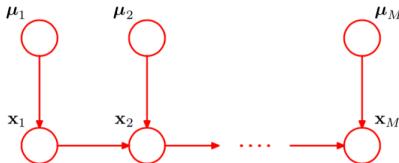


Figure 8.12 As in Figure 8.11 but with a single set of parameters μ shared amongst all of the conditional distributions $p(x_i|x_{i-1})$.

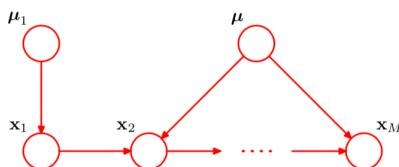


図: 1-3 グラフィカルモデルの例

図 1-3 は PRML8 章からの抜粋ですが、このようにモデリングにおいてもグラフは利用されています。

今回はグラフィカルモデルのメリットと言うことで、8 章の冒頭部から引用します。

グラフィカルモデルは以下のような便利な特徴を持っている。

1. 確率モデルの構造を視覚化する簡単な方法を提供し、新しいモデルの設計方針を決めるのに役立つ
2. グラフの構造を調べることにより、条件付き独立性などのモデルの性質に関する知見が得られる
3. 精巧なモデルにおいて推論や学習を実行するためには複雑な計算が必要となるが、これを数学的な表現を暗に伴うグラフ上の操作として表現することができる

上記がグラフィカルモデルの利点として言及されていました。このようにグラフ理論は複雑な概念を視覚化する手段として用いることで大きな武器となります。精巧で複雑なモデリングを行えば行うほど状況の整理が大変になるので、グラフィカルに表現すると良いです。

PRML8章では有効グラフィカルモデル(directed graphical modeling)の例としてベイジアンネットワーク(Bayesian network)、無向グラフとしてマルコフ確率場(Markov random field)について言及されています。

1.2.2 TensorFlow と計算グラフ

1-2-1節でグラフィカルモデルの利点をまとめたのですが、DeepLearningで用いられているライブラリでこちらをうまく利用しているのがTensorFlowです。TensorFlowではデータフローグラフという概念が用いられています。

一連の計算における数学的な演算ノード、データの配列をエッジとして表現するグラフとして表すことで、複雑なモデリングも直感的に取り扱うことができます。

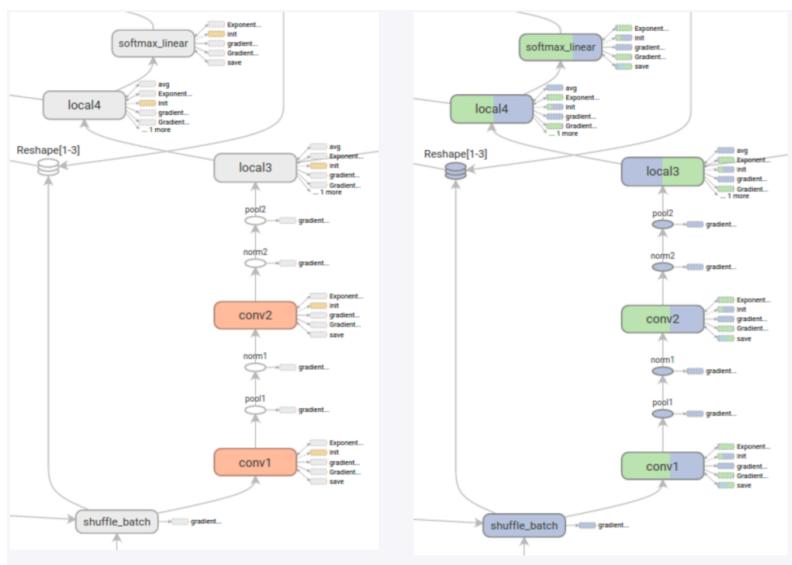


図: 1-4 TensorBoard による演算の可視化

上記のようにグラフとして扱った演算における計算の流れを TensorBoard を用いて可視化を行うことも可能です。複雑な演算を行う際のコーディングは非常に不安になるものですが、このように可視化を行うことでミスを減らすことができます。

1.3 Survey の概要と数式表記

1-3 節では本書において主に参考にする Survey やその数式的な表記についてご紹介します。以下、1-3-1 では Survey の概要と目次について、1-3-2 節では Survey の Section2-B に記されている数式表記についてご紹介します。

1.3.1 Survey の概要と目次

1-3-1 節では Survey の概要と目次についてご紹介します。

JOURNAL OF LATEX CLASS FILES, VOL. XX, NO. XX, AUGUST 2019

A Comprehensive Survey on Graph Neural Networks

Zonghan Wu, Shirui Pan, *Member, IEEE*, Fengwen Chen, Guodong Long,
Chengqi Zhang, *Senior Member, IEEE*, Philip S. Yu, *Fellow, IEEE*

Abstract—Deep learning has revolutionized many machine learning tasks in recent years, ranging from image classification and video processing to speech recognition and natural language understanding. The data in these tasks are typically represented in the Euclidean space. However, there is an increasing number of applications where data are generated from non-Euclidean domains and are represented as graphs with complex relationships and interdependency between objects. The complexity of graph data has imposed significant challenges on existing machine learning algorithms. Despite many studies on extending deep learning to graphs, there is still a lack of comprehensive surveys on this topic.

example, we can represent an image as a regular grid in the Euclidean space. A convolutional neural network (CNN) is able to exploit the shift-invariance, local connectivity, and compositionality of image data [9]. As a result, CNNs can extract local meaningful features that are shared with the entire data sets for various image analysis.

While deep learning effectively captures hidden patterns of Euclidean data, there is an increasing number of applications

図: 1-5 A Comprehensive Survey on GNNs

<https://arxiv.org/abs/1901.00596>

本書では上記の Survey の "A Comprehensive Survey on Graph Neural Networks" を主に参考にします。

I Introduction
▼ II Background & Definition
II-A Background
II-B Definition
▼ III Categorization and Frameworks
III-A Taxonomy of Graph Neural Networks (GNNs)
III-B Frameworks
IV Recurrent Graph Neural Networks
▼ V Convolutional Graph Neural Networks
V-A Spectral-based ConvGNNs
V-B Spatial-based ConvGNNs
V-C Graph Pooling Modules
V-D Discussion of Theoretical Aspects
▼ VI Graph autoencoders
VI-A Network Embedding
VI-B Graph Generation
VII Spatial-temporal Graph Neural Networks
▼ VIII Applications
VIII-A Data Sets
VIII-B Evaluation & Open-source Implementations
VIII-C Practical Applications
IX Future Directions
X Conclusion

図: 1-6 Survey の目次

まず、Survey の目次は上記となっています。研究トピックについてはIVの Recurrent Graph Neural Networks からVIIの Spatial-temporal Graph Neural Networks までをそれぞれ整理したとされています。こちらを確認した上で筆者の見解として、IVとVが手法的な話、VIとVIIがタスクの話と読み替える方がわかりやすいと思われたので、それぞれ第2章と第3章のトピックとしました。

次に、II-B では Definition ということで、Survey 全体を通しての数式定義

義について記載されていたので、こちらについては実際に本論に入る前に1-3-2節で確認することとしました。さらに本論に入る前にGraph Neural Networksについての概観も行えればということで、1-4節ではSurveyのAbstractの内容を元にGraph Neural Networksについて簡単に確認を行っています。

また、VIIIのApplicationsではOSSなどについても紹介されており、中でもDGL(Deep Graph Library)というライブラリが便利と記述されているように思われたため、第4章でDGLの紹介や動作確認などが行えるようにいたしました。

1.3.2 グラフ理論の数式的な定義

1-3-2節ではSurveyのSection2-BのDefinitionより、グラフ理論の数式的な定義について取り扱います。

B. Definition

Throughout this paper, we use bold uppercase characters to denote matrices and bold lowercase characters denote vectors. Unless particularly specified, the notations used in this paper are illustrated in Table I. Now we define the minimal set of definitions required to understand this paper.

図: 1-7 冒頭部 (Section2-B)

冒頭部の記述では、論文を通しての記法の注意として太字の大文字は行列(matrices)を表し、太字の小文字はベクトルを表すとされています。それぞれの詳細についてはTable1にまとめたとされています。

TABLE I: Commonly used notations.

Notations	Descriptions
$ \cdot $	The length of a set.
\odot	Element-wise product.
G	A graph.
V	The set of nodes in a graph.
v	A node $v \in V$.
E	The set of edges in a graph.
e_{ij}	An edge $e_{ij} \in E$.
$N(v)$	The neighbors of a node v .
\mathbf{A}	The graph adjacency matrix.
\mathbf{A}^T	The transpose of the matrix \mathbf{A} .
$\mathbf{A}^n, n \in \mathbb{Z}$	The n^{th} power of \mathbf{A} .
$[\mathbf{A}, \mathbf{B}]$	The concatenation of \mathbf{A} and \mathbf{B} .
\mathbf{D}	The degree matrix of \mathbf{A} . $\mathbf{D}_{ii} = \sum_{j=1}^n \mathbf{A}_{ij}$.
n	The number of nodes, $n = V $.
m	The number of edges, $m = E $.
d	The dimension of a node feature vector.
b	The dimension of a hidden node feature vector.
c	The dimension of an edge feature vector.
$\mathbf{X} \in \mathbf{R}^{n \times d}$	The feature matrix of a graph.
$\mathbf{x} \in \mathbf{R}^n$	The feature vector of a graph in the case of $d = 1$.
$\mathbf{x}_v \in \mathbf{R}^d$	The feature vector of the node v .
$\mathbf{X}^e \in \mathbf{R}^{m \times c}$	The edge feature matrix of a graph.
$\mathbf{x}_{(v,u)}^e \in \mathbf{R}^c$	The edge feature vector of the edge (v, u) .
$\mathbf{X}^{(t)} \in \mathbf{R}^{n \times d}$	The node feature matrix of a graph at the time step t .
$\mathbf{H} \in \mathbf{R}^{n \times b}$	The node hidden feature matrix.
$\mathbf{h}_v \in \mathbf{R}^b$	The hidden feature vector of node v .
k	The layer index.
t	The time step/iteration index.
$\sigma(\cdot)$	The sigmoid activation function.
$\sigma_h(\cdot)$	The tangent hyperbolic activation function.
$\mathbf{W}, \Theta, w, \theta$	Learnable model parameters.

図: 1-8 Table1

Table1 には上記のように様々な記法についてまとめられています。それぞれの説明については Definition1～Definition3 に記載されているので、そちらを確認していきます。

Definition 1 (Graph): A graph is represented as $G = (V, E)$ where V is the set of vertices or nodes (we will use nodes throughout the paper), and E is the set of edges. Let $v_i \in V$ to denote a node and $e_{ij} = (v_i, v_j) \in E$ to denote an edge pointing from v_j to v_i . The neighborhood of a node v is defined as $N(v) = \{u \in V | (v, u) \in E\}$. The adjacency matrix A is a $n \times n$ matrix with $A_{ij} = 1$ if $e_{ij} \in E$ and $A_{ij} = 0$ if $e_{ij} \notin E$. A graph may have node attributes \mathbf{X} , where $\mathbf{X} \in \mathbf{R}^{n \times d}$ is a node feature matrix with $\mathbf{x}_v \in \mathbf{R}^d$ representing the feature vector of a node v . Meanwhile, a graph may have edge attributes \mathbf{X}^e , where $\mathbf{X}^e \in \mathbf{R}^{m \times c}$ is an edge feature matrix with $\mathbf{x}_{v,u}^e \in \mathbf{R}^c$ representing the feature vector of an edge (v, u) .

図: 1-9 Definition1(Section2-B)

1
9

Definition1 ではグラフ (Graph) に関する定義がまとめられています。グラフは $G = (V, E)$ のように表し、 V はノードの集合 (V は Vertice からとっています)、 E はエッジの集合を表すとされています。また、ノードを $v_i \in V$ 、エッジを $e_{ij} = (v_i, v_j) \in E$ と表しています。次にノードの近傍 (neighborhood) のノードを $N(v) = \{u \in V | (v, u) \in E\}$ 、隣接行列 (adjacency matrix) を A と表すとされています。またノードの属性を $\mathbf{X} \in \mathbf{R}^{n \times d}$ 、エッジの属性を $\mathbf{X}^e \in \mathbf{R}^{m \times c}$ と表すとなっています。ここで注意すべきは、ノードとエッジの属性は 1 つとは限らず複数ある場合もあるという点です。ここでは一つのノードあたり d 個の属性、一つのエッジ (二つのノードのペアに対するエッジ) は c 個の属性を持つと考えられます。また、この時 n はノードの個数、 m はエッジの個数であるとされています。

Definition 2 (Directed Graph): A directed graph is a graph with all edges directed from one node to another. An undirected graph is considered as a special case of directed graphs where there is a pair of edges with inverse directions if two nodes are connected. A graph is undirected if and only if the adjacency matrix is symmetric.

図: 1-10 Definition2(Section2-B)

Definition2 では有向グラフ (Directed Graph) についてまとめられています。無向グラフ (Undirected Graph) は有向グラフの特殊なケースで、隣接行列が対称 (symmetric) であるとき無向グラフであるとされています。

Definition 3 (Spatial-Temporal Graph): A spatial-temporal graph is an attributed graph where the node attributes change dynamically over time. The spatial-temporal graph is defined as $G^{(t)} = (\mathbf{V}, \mathbf{E}, \mathbf{X}^{(t)})$ with $\mathbf{X}^{(t)} \in \mathbf{R}^{n \times d}$.

図: 1-11 Definition3(Section2-B)

Definition3 ではノードの属性の \mathbf{X} が時間によって変動するグラフのことを Spatial-Temporal Graph であると定義しています。

1.4 Graph Neural Networks

1-4 節では、第2章、第3章で本論として研究トピックについて見ていく前に、Survey の Abstract の和訳と解説を通して Graph Neural Networks についての大枠な把握を行います。1-4-1 節では背景の話にあたる内容について、1-4-2 節では Graph Neural Networks の研究トピックの紹介について行います。

1.4.1 背景 (Background)

Deep learning has revolutionized many machine learning tasks in recent years, ranging from image classification and video processing to speech recognition and natural language understanding.

和訳：『DeepLearning は近年機械学習関連のタスクにおいて多大なる結果を残しており、その応用範囲は画像識別 (image classification) や動画処理 (video processing) から音声識別 (speech recognition) や言語理解 (natural language understanding) にまで広がっている。』

解説：『様々な分野における DeepLearning の成功について記述されています。』

The data in these tasks are typically represented in the Euclidean space. However, there is an increasing number of applications where data are generated from non-Euclidean domains and are represented as graphs with complex relationships and interdependency between objects.

和訳：『(前文で挙げた) タスクにおけるデータはユークリッド空間 (Euclidean space) 上で表現できる。しかしながら、世の中にはオブジェクト間の関係性や独立性を取り扱うグラフのような非ユークリッド領域上のデータを用いる数多くの応用 (applications) がある。』

若干訳しづらい内容でしたので意訳も入りましたがそこまで大きな違いはないかと思います。グラフのデータは画像のように綺麗なグリッド (網目) 状であるわけではなく、イレギュラーな構造であるということについて示唆されています。

The complexity of graph data has imposed significant challenges on existing machine learning algorithms. Recently, many studies on

extending deep learning approaches for graph data have emerged. In this survey, we provide a comprehensive overview of graph neural networks (GNNs) in data mining and machine learning fields.

和訳：『グラフのデータの複雑さは既存の機械学習のアルゴリズムにおける大きな課題となっている。最近では DeepLearning を用いてグラフデータを取り扱うアプローチについて多くの研究がなされている。このサーベイ (論文) では、データマイニングや機械学習の分野における GNN(Graph Neural Networks) の包括的な大枠を提示する。』

DeepLearning を用いたグラフのデータへのアプローチが多くなされるようになってきたため、サーベイを行ったことについて記載がされています。

=====

ここまで記述で、非ユークリッド空間について取り扱った Graph data への DeepLearning の適用にあたって、これまで様々なアプローチが取られてきたとされています。これらの応用や手法について Survey ではまとめており、第2章以降で詳しく確認します。

2
2

1.4.2 GNN の研究トピックの紹介

We propose a new taxonomy to divide the state-of-the-art graph neural networks into four categories, namely recurrent graph neural networks, convolutional graph neural networks, graph autoencoders, and spatial-temporal graph neural networks.

和訳：『我々は SotA のグラフニューラルネットワークを 4 つに分ける新しい分類を提案する。4 つの分類としては、再帰的グラフニューラルネットワーク (recurrent graph neural networks)、畳み込みグラフニューラルネットワーク (convolutional neural networks)、グラフオートエンコーダ (graph autoencoders)、spatial-temporal graph neural networks の 4 つである。』

分類を示してくれているので、研究の内容を整理しやすくなっています。この4つの分類については念頭に置いた上で読み進めていくと良さそうです。

We further discuss the applications of graph neural networks across various domains and summarize the open source codes, benchmark data sets, and model evaluation of graph neural networks. Finally, we propose potential research directions in this rapidly growing field.

和訳：『さらに我々は様々な領域におけるグラフニューラルネットワークの応用について議論し、オープンソースのコードやベンチマークのデータセットやグラフニューラルネットワークのモデルの評価をまとめた。最終的に我々はこの急速に発展する領域における研究の方向性の可能性についてまとめた。』

研究を整理した上で将来研究の可能性についてまとめられており、ざっくり概要を掴むにはちょうど良い内容となっているようです。

=====

Graph Neural Networks の主な研究トピックとして、再帰的グラフニューラルネットワーク (RecGNNs; recurrent graph neural networks)、畳み込みグラフニューラルネットワーク (ConvGNNs; convolutional neural networks)、グラフオートエンコーダ (GAEs; graph autoencoders)、spatial-temporal graph neural networks(STGNNs) の4つが挙げられています。

本書では4つある中の前半2つを手法、後半2つを応用タスクとみなすことで、よりそれぞれの位置づけが理解しやすくなるようにしています。詳細の記載では、GAEs と STGNNs では RecRNNs や ConvGNNs を用いるなどの記述が出てくるため混乱を防ぐためでもあります。前半2つのトピックを第2章、後半2つのトピックを第3章として取り扱っています。

また、OSS やベンチマークなどについても言及されています。中でも実際に実装して動かすことのできるライブラリについては非常に興味深いた

め、紹介されているライブラリの DGL(Deep Graph Library)について第4章で動かしながら確認していきます。

1.5 第1章のまとめ

第2章以降で詳細の内容に入る所以、簡単に第1章の内容をまとめておきます。第1章ではグラフ理論の基本的な内容や応用例、Graph Neural Networksについて取り扱いました。

グラフ理論について初めて触れた方などは、ノードとエッジがあり、それぞれが属性を持ち、それをニューラルネットワークを用いて何かしらのタスクを解いていくということについて改めて抑えていただければ一旦は十分だと思います。

第2章

Graph Neural Networks の 手法

25

第2章では、Graph Neural Networksの手法について取り扱います。Graph Neural Networksは大別すると RecGNNs(Recurrent Graph Neural Networks)とConvGNNs(Convolutional Graph Neural Networks)の二つに分けることができます。それぞれ、2-1節で RecGNNs、2-2節で ConvGNNsについて取り扱います。

2.1 Recurrent GNNs

2-1節ではSurveyのSection4のRecurrent graph neural networksについて取り扱います。

IV. RECURRENT GRAPH NEURAL NETWORKS

Recurrent graph neural networks (RecGNNs) are mostly pioneer works of GNNs. They apply the same set of parameters recurrently over nodes in a graph to extract high-level node representations. Constrained by computational power, earlier research mainly focused on directed acyclic graphs [13], [80].

図: 2-1 1st パラグラフ (Section4)

Survey の Section4 の冒頭部では、RecGNNs(Recurrent graph neural networks) が GNNs の初期のエポックメイキングな研究であるということに触れた上で、初期の研究ではマシンのリソースの制約で有向非巡回グラフ (DAG; Directed Acyclic Graphs) にフォーカスしていたことについて言及されています。

有向非巡回グラフ

出典: フリー百科事典『ウィキペディア (Wikipedia)』

有向非巡回グラフ、**有向非循環グラフ**、**有向無閉路グラフ**（ゆうこうひじゅんかいグラフ、英: Directed acyclic graph, DAG）とは、グラフ理論における閉路のない有向グラフのことである。有向グラフは頂点と有向辺（方向を示す矢印付きの辺）からなり、辺は頂点同士をつなぐが、ある頂点 v から出発し、辺をたどり、頂点 v に戻ってこないのが有向非巡回グラフである[1][2][3]。

有向非巡回グラフは様々な情報をモデル化するのに使われる。有向非巡回グラフにおける到達可能性は半順序を構成し、全ての有限半順序は到達可能性を利用し有向非巡回グラフで表現可能である。順序づけする必要があるタスクの集合は、あるタスクが他のタスクよりも前に行う必要があるという制約により、頂点をタスク、辺を制約条件で表現する有向非巡回グラフで表現できる。トポロジカルソートを使うと、妥当な順序を手に入れることができる。加えて、有向非巡回グラフは一部が重なるシーケンスの集合を表現する際の空間効率の良い表現として利用できる。また、有向非巡回グラフはイベント間の因果関係を表現することにも使える。さらに、有向非巡回グラフはデータの流れが一定方向のネットワークを表現することにも使える。

図: 2-2 有向非巡回グラフ (DAG; Directed Acyclic Graph)

<https://ja.wikipedia.org/wiki/有向非巡回グラフ>

有向非巡回グラフは、上記のように閉路を持たない有向グラフで、ある頂点 v から出発した時に v に戻ってこないグラフであるとされています。

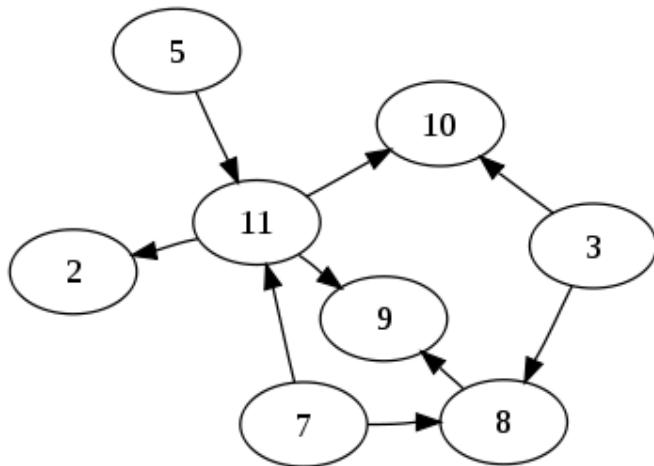


図: 2-3 有向非巡回グラフの図

また、上記は Wikipedia の記事の図をお借りしましたが、全てのノードが一度他のノードに推移すると自身に戻ってくることがないことが確認できます。また、全ての矢印は最終的に 2 と 10 のノードにたどり着くようになっていることも確認できます。このような性質を使うことで有効非巡回グラフは時間ステップに関連するモデリングに使うことができます。

Graph Neural Network (GNN*) proposed by Scarselli et al. extends prior recurrent models to handle general types of graphs, e.g., acyclic, cyclic, directed, and undirected graphs [15]. Based on an information diffusion mechanism, GNN* updates nodes' states by exchanging neighborhood information recurrently until a stable equilibrium is reached. A node's hidden state is recurrently updated by

$$\mathbf{h}_v^{(t)} = \sum_{u \in N(v)} f(\mathbf{x}_v, \mathbf{x}_{e(v,u)}, \mathbf{x}_u, \mathbf{h}_u^{(t-1)}), \quad (1)$$

where $f(\cdot)$ is a parametric function, and $\mathbf{h}_v^{(0)}$ is initialized randomly. The sum operation enables GNN* to be applicable to all nodes, even if the number of neighbors differs and no neighborhood ordering is known. To ensure convergence, the recurrent function $f(\cdot)$ must be a contraction mapping, which shrinks the distance between two points after projecting them into a latent space. In the case of $f(\cdot)$ being a neural network, a penalty term has to be imposed on the Jacobian matrix of parameters. When a convergence criterion is satisfied, the last step node hidden states are forwarded to a readout layer. GNN* alternates the stage of node state propagation and the stage of parameter gradient computation to minimize a training objective. This strategy enables GNN* to handle cyclic graphs. In follow-up works, Graph Echo State Network (GraphESN) [16] extends echo state networks to improve the

training efficiency of GNN*. GraphESN consists of an encoder and an output layer. The encoder is randomly initialized and requires no training. It implements a contractive state transition function to recurrently update node states until the global graph state reaches convergence. Afterward, the output layer is trained by taking the fixed node states as inputs.

図: 2-4 2nd パラグラフ (Section4)

第二パラグラフでは、Scarselli などによって提案された GNN*(Graph

Neural Network) は、先行研究における recurrent なモデルが有向非巡回グラフ (DAG) 前提だったのに対し、無向グラフや巡回型のグラフにも適用できるようになったとされています。その他の Graph Neural Networks の研究と区別するために、ここでは GNN* と表記しているようです。GNN* では安定的な均衡状態に到達するまで、ノードが再帰的に近傍のノードと情報交換し、自身の状態をアップデートすることについて記載されています。また、この処理の流れを数式 (1) で表現されています。エッジやノードの属性と、ステップ $t-1$ における $h_u^{(t-1)}$ を用いることで次ステップの t における $h_v^{(t)}$ を計算しています。

Gated Graph Neural Network (GGNN) [17] employs a gated recurrent unit (GRU) [81] as a recurrent function, reducing the recurrence to a fixed number of steps. The advantage is that it no longer needs to constrain parameters to ensure convergence. A node hidden state is updated by its previous hidden states and its neighboring hidden states, defined as

$$\mathbf{h}_v^{(t)} = \text{GRU}(\mathbf{h}_v^{(t-1)}, \sum_{u \in N(v)} \mathbf{W} \mathbf{h}_u^{(t-1)}), \quad (2)$$

where $\mathbf{h}_v^{(0)} = \mathbf{x}_v$. Different from GNN* and GraphESN, GGNN uses the back-propagation through time (BPTT) algorithm to learn the model parameters. This can be problematic for large graphs, as GGNN needs to run the recurrent function multiple times over all nodes, requiring the intermediate states of all nodes to be stored in memory.

図: 2-5 3rd パラグラフ (Section4)

第三パラグラフでは、GRU(Gated Recurrent Unit) を導入した GGNN(Gated Graph Neural Network) について記載されています。

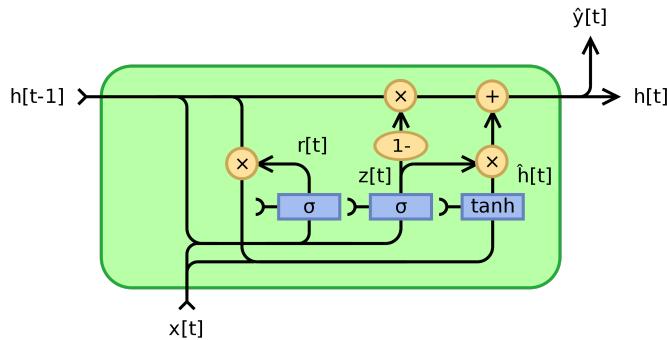


図: 2-6 GRU Unit

GRU は上記のような Unit で LSTM に似ています。こちらを用いることで数式 (2) のように $h_v^{(t)}$ の計算を行なっています。GNN*などと違い、GGNN は BPTT(back-propagation through time) アルゴリズムを用いてモデルのパラメータを学習させるとされています。とはいっても、この GGNN は大きなグラフに適用するにはメモリ保持を必要とするという問題点があると言及されています。

Stochastic Steady-state Embedding (SSE) proposes a learning algorithm that is more scalable to large graphs [18]. SSE updates node hidden states recurrently in a stochastic and asynchronous fashion. It alternatively samples a batch of nodes for state update and a batch of nodes for gradient computation. To maintain stability, the recurrent function of SSE is defined as a weighted average of the historical states and new states, which takes the form

$$\mathbf{h}_v^{(t)} = (1 - \alpha)\mathbf{h}_v^{(t-1)} + \alpha \mathbf{W}_1 \sigma(\mathbf{W}_2[\mathbf{x}_v, \sum_{u \in N(v)} [\mathbf{h}_u^{(t-1)}, \mathbf{x}_u]]), \quad (3)$$

where α is a hyper-parameter, and $\mathbf{h}_v^{(0)}$ is initialized randomly. While conceptually important, SSE does not theoretically prove that the node states will gradually converge to fixed points by applying Equation 3 repeatedly.

図: 2-7 4th パラグラフ (Section4)

第四パラグラフでは、大きなグラフを取り扱う上での GGNN の課題を解決するにあたって提案された、SSE(Steady-state Embedding) が紹介されています。SSE ではノードの隠れ層のアップデートを確率的かつ非同期な流れで、パラメータのアップデートを行うとされています。安定性も考慮し、数式 (3) のようにハイパー-パラメータの α を用いて一部情報の保持ができるようにアップデートを行なっています。

2.2 Convolutional GNNs

2-2 節では Survey の Section5 の Convolutional Graph Neural Networks について取り扱います。

V. CONVOLUTIONAL GRAPH NEURAL NETWORKS

Convolutional graph neural networks (ConvGNNs) are closely related to recurrent graph neural networks. Instead of iterating node states with contractive constraints, ConvGNNs address the cyclic mutual dependencies architecturally using a fixed number of layers with different weights in each layer. This key distinction is illustrated in Figure 3. As graph convolutions are more efficient and convenient to composite with other neural networks, the popularity of ConvGNNs has been rapidly growing in recent years. ConvGNNs fall into two categories, spectral-based and spatial-based. Spectral-based approaches define graph convolutions by introducing filters from the perspective of graph signal processing [82] where the graph convolutional operation is interpreted as removing noises from graph signals. Spatial-based approaches inherit ideas from RecGNNs to define graph convolutions by information propagation. Since GCN [22] bridged the gap between spectral-based approaches and spatial-based approaches, spatial-based methods have developed rapidly recently due to its attractive efficiency, flexibility, and generality.

3
2

図: 2-8 冒頭部 (Section4)

冒頭部では、ConvGNNs(Convolutional Graph Neural Network)とRecGNNsとの関連について記載されており、ConvGNNsの形式が他のニューラルネットワークと複合するにあたって効率的かつ便利であったため、ConvGNNsの利用が近年非常に増えてきているとされています。ConvGNNsはspectral-basedとspatial-basedの二つのカテゴリに分けることができるとされています。spectral-basedがSection5-A、spatial-basedがSection5-Bでそれぞれ詳しく記述されているため、それぞれ2-2-1節、2-2-2節で確認して行きます。

2.2.1 Spectral-based ConvGNNs(Section5-A)

2-2-1 節では Section5-A の Spectral-based ConvGNNs について取り扱います。

A. Spectral-based ConvGNNs

Background Spectral-based methods have a solid mathematical foundation in graph signal processing [82], [83], [84]. They assume graphs to be undirected. The normalized graph Laplacian matrix is a mathematical representation of an undirected graph, defined as $\mathbf{L} = \mathbf{I}_n - \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}}$, where \mathbf{D} is a diagonal matrix of node degrees, $\mathbf{D}_{ii} = \sum_j (\mathbf{A}_{i,j})$. The normalized graph Laplacian matrix possesses the property of being real symmetric positive semidefinite. With this property, the normalized Laplacian matrix can be factored as

$\mathbf{L} = \mathbf{U} \Lambda \mathbf{U}^T$, where $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \dots, \mathbf{u}_{n-1}] \in \mathbb{R}^{n \times n}$ is the matrix of eigenvectors ordered by eigenvalues and Λ is the diagonal matrix of eigenvalues (spectrum), $\Lambda_{ii} = \lambda_i$.

図: 2-9 1st パラグラフ _前半 (Section5-A)

第一パラグラフの前半では、Spectral-based な手法は graph signal processing に基づいていることについて言及し、いくつかの数式を用いて問題を定義しています。ここで正規化ラプラシアン行列 (normalized graph Laplacian matrix) の \mathbf{L} を導入し、ここで用いられている \mathbf{D} はノードの次数の対角行列 (diagonal matrix)、 \mathbf{A} は隣接行列 (Adjacency matrix)、 \mathbf{I}_n は単位行列を表しています。ここで \mathbf{L} の行列分解を考え、 $\mathbf{L} = \mathbf{U} \Lambda \mathbf{U}^T$ のように書いています。ここで Λ は \mathbf{L} の固有値の対角ベクトル (diagonal matrix of eigenvalues) を表すとされています。

The eigenvectors of the normalized Laplacian matrix form an orthonormal space, in mathematical words $\mathbf{U}^T \mathbf{U} = \mathbf{I}$. In graph signal processing, a graph signal $\mathbf{x} \in \mathbf{R}^n$ is a feature vector of all nodes of a graph where x_i is the value of the i^{th} node. The *graph Fourier transform* to a signal \mathbf{x} is defined as $\mathcal{F}(\mathbf{x}) = \mathbf{U}^T \mathbf{x}$, and the inverse graph Fourier transform is defined as $\mathcal{F}^{-1}(\hat{\mathbf{x}}) = \mathbf{U}\hat{\mathbf{x}}$, where $\hat{\mathbf{x}}$ represents the resulted signal from the graph Fourier transform. The graph Fourier transform projects the input graph signal to the orthonormal space where the basis is formed by eigenvectors of the normalized graph Laplacian. Elements of the transformed signal $\hat{\mathbf{x}}$ are the coordinates of the graph signal in the new space so that the input signal can be represented as $\mathbf{x} = \sum_i \hat{x}_i \mathbf{u}_i$, which is exactly the inverse graph Fourier transform. Now the graph convolution of the input signal \mathbf{x} with a filter $\mathbf{g} \in \mathbf{R}^n$ is defined as

$$\begin{aligned} \mathbf{x} *_G \mathbf{g} &= \mathcal{F}^{-1}(\mathcal{F}(\mathbf{x}) \odot \mathcal{F}(\mathbf{g})) \\ &= \mathbf{U}(\mathbf{U}^T \mathbf{x} \odot \mathbf{U}^T \mathbf{g}), \end{aligned} \quad (4)$$

where \odot denotes the element-wise product. If we denote a filter as $\mathbf{g}_\theta = \text{diag}(\mathbf{U}^T \mathbf{g})$, then the spectral graph convolution is simplified as

$$\mathbf{x} *_G \mathbf{g}_\theta = \mathbf{U} \mathbf{g}_\theta \mathbf{U}^T \mathbf{x}. \quad (5)$$

Spectral-based ConvGNNs all follow this definition. The key difference lies in the choice of the filter \mathbf{g}_θ .

図: 2-10 1st パラグラフ_後半 (Section5-A)

第一パラグラフの後半では、シグナル \mathbf{x} に対するグラフフーリエ変換 (graph Fourier transform) として $F(\mathbf{x}) = \mathbf{U}^T \mathbf{x}$ を定義し、また逆フーリエ変換 (inverse graph Fourier transform) として $F^{-1}(\hat{\mathbf{x}}) = \mathbf{U}\hat{\mathbf{x}}$ を定義しています。この時、 $\hat{\mathbf{x}}$ は $F(\mathbf{x})$ の出力を意味しているとされています。この時グラフの畠み込み (graph convolution) は数式 (4) や (5) で表すことができるとしています。

2.2.2 Spatial-based ConvGNNs(Section5-B)

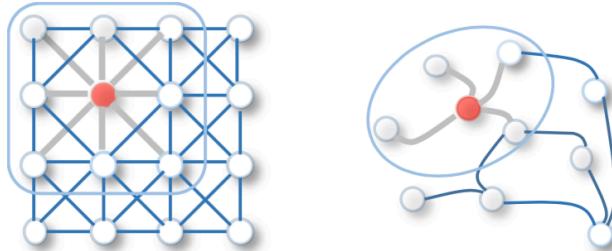
2-2-2 節では Section5-B の Spatial-based ConvGNNs について取り扱います。

B. Spatial-based ConvGNNs

Analogous to the convolutional operation of a conventional CNN on an image, spatial-based methods define graph convolutions based on a node's spatial relations. Images can be considered as a special form of graph with each pixel representing a node. Each pixel is directly connected to its nearby pixels, as illustrated in Figure 1a. A filter is applied to a 3×3 patch by taking the weighted average of pixel values of the central node and its neighbors across each channel. Similarly, the spatial-based graph convolutions convolve the central node's representation with its neighbors' representations to derive the updated representation for the central node, as illustrated in Figure 1b. From another perspective, spatial-based ConvGNNs share the same idea of information propagation/message passing with RecGNNs. The spatial graph convolutional operation essentially propagates node information along edges.

図: 2-11 1st パラグラフ (Section5-B)

第一パラグラフでは、画像などにおける従来の CNN の演算 (convolutional operation of a conventional CNN) と同様に spatial-based の演算を定義するとしています。



(a) 2D Convolution. Analogous to a graph, each pixel in an image is taken as a node where neighbors are determined by the filter size. The 2D convolution takes the weighted average of pixel values of the red node along with its neighbors. The neighbors of a node are ordered and have a fixed size.

(b) Graph Convolution. To get a hidden representation of the red node, one simple solution of the graph convolutional operation is to take the average value of the node features of the red node along with its neighbors. Different from image data, the neighbors of a node are unordered and variable in size.

Fig. 1: 2D Convolution vs. Graph Convolution.

3
6

図: 2-12 2D Conv vs. Graph Conv

上記の Figure1 で概念的なイメージが記載されています。左が画像に対する畳み込みで画像などに用いられる通常の CNN などで使われており、右が Graph Convolution として定義されています。ここでは、通常の畳み込みは近傍の値 (画像の場合は近傍のピクセルの値) に基づいて計算する一方で、グラフ畳み込みは近傍をノードのインデックス (Euclidean) ではなく、エッジで連結されているノードを近傍と見なす (non-Euclidean) ことに注意が必要です。

Neural Network for Graphs (NN4G) [24], proposed in parallel with GNN*, is the first work towards spatial-based ConvGNNs. Distinctively different from RecGNNs, NN4G learns graph mutual dependency through a compositional neural architecture with independent parameters at each layer. The neighborhood of a node can be extended through incremental construction of the architecture. NN4G performs graph convolutions by summing up a node's neighborhood information directly. It also applies residual connections and skip connections to memorize information over each layer. As a result, NN4G derives its next layer node states by

$$\mathbf{h}_v^{(k)} = f(\mathbf{W}^{(k)^T} \mathbf{x}_v + \sum_{i=1}^{k-1} \sum_{u \in N(v)} \Theta^{(k)^T} \mathbf{h}_u^{(k-1)}), \quad (15)$$

where $f(\cdot)$ is an activation function and $\mathbf{h}_v^{(0)} = \mathbf{0}$. Equation 15 can also be written in a matrix form:

$$\mathbf{H}^{(k)} = f(\mathbf{X}\mathbf{W}^{(k)} + \sum_{i=1}^{k-1} \mathbf{A}\mathbf{H}^{(k-1)}\Theta^{(k)}), \quad (16)$$

which resembles the form of GCN [22]. One difference is that NN4G uses the unnormalized adjacency matrix which may potentially cause hidden node states to have extremely different scales. Contextual Graph Markov Model (CGMM) [47] proposes a probabilistic model inspired by NN4G. While maintaining spatial locality, CGMM has the benefit of probabilistic interpretability.

図: 2-13 2nd パラグラフ (Section5-B)

第二パラグラフでは RecGNNs の GNN*と並行で提案された、NN4G(Neural Network for Graphs) が spatial-based ConvGNNs の最初の研究であるとされています。畠み込みの演算としては数式 (15) で記載されており、後ろの項でパラメータの θ を用いて隠れ層の値の畠み込みを行なっています。また、数式 (16) は数式 (15) を行列の形式で書き直したものであるとされています。

2.2.3 Graph Pooling Modules(Section5-C)

2-2-3 節では Section5-C の Graph Pooling Modules について取り扱います。

Nowadays, mean/max/sum pooling is the most primitive and effective way to implement down-sampling since calculating

the mean/max/sum value in the pooling window is fast:

$$\mathbf{h}_G = \text{mean/max/sum}(\mathbf{h}_1^{(K)}, \mathbf{h}_2^{(K)}, \dots, \mathbf{h}_n^{(K)}), \quad (27)$$

where K is the index of the last graph convolutional layer.

3
8

図: 2-14 Section5-C の記載

入力した graph data の down sampling を行うにあたって pooling を行うとされており、pooling に置いて一番シンプルな方法として平均 (mean)/最大値 (max)/合計 (sum) などがあげられています。ここで K はレイヤーのインデックスを表しているとされています。

2.2.4 Discussion of Theoretical Aspects(Section5-D)

追記するかしないかは第 2 版時に判断します。

第3章では、Graph Neural Networks のタスク定義について取り扱います。AutoEncoder に GNNs を導入した GAEs(Graph AutoEncoders) と、空間的時間的な依存関係について取り扱った STGNNs(Spatial-Temporal Graph Neural Networks) についてご紹介します。それぞれ 3-1 節で GAEs、3-2 節で STGNNs についてご紹介します。

3.1 GAEs(Graph AutoEncoders)

3-1 節では Survey の Sectio6 の Graph AutoEncoders について取り扱います。

VI. GRAPH AUTOENCODERS

Graph autoencoders (GAEs) are deep neural architectures which map nodes into a latent feature space and decode graph information from latent representations. GAEs can be used to learn network embeddings or generate new graphs. The main characteristics of selected GAEs are summarized in Table V. In the following, we provide a brief review of GAEs from two perspectives, network embedding and graph generation.

図: 3-1 冒頭部 (Section6)

冒頭部では、GAEs(Graph AutoEncoders)の概要として、ノード群を潜在的な特徴量空間 (latent feature space) に写像する深層ニューラルネットワークの構造で (encoder)、潜在表現 (latent representation) から decode してグラフの情報を得るとされています。GAEs は network embedding の学習や新しいグラフの生成に用いることができるとされており、Network Embedding が Section6-A、Graph Generation が Section6-B で記述されています。そのため、Network Embedding を 1-1 節、Graph Generation を 1-2 節でそれぞれ取り扱います。

3.1.1 Network Embedding(Section6-A)

3-1-1 節では Section6-A の Network Embedding について取り扱います。

A. Network Embedding

A network embedding is a low-dimensional vector representation of a node which preserves a node's topological information. GAEs learn network embeddings using an encoder to extract network embeddings and using a decoder to enforce network embeddings to preserve the graph topological information such as the PPMI matrix and the adjacency matrix.

図: 3-2 1st パラグラフ (Section6-A)

第一パラグラフでは、Network Embedding の概要として、node の位相的な (topological) 情報を保存する低次元のベクトル表現 (low-dimensional vector representation) であると述べています。また、encoder がベクトル表現をグラフから抽出し、decoder では反対にベクトル表現からグラフを生成するとされています。

Earlier approaches mainly employ multi-layer perceptrons to build GAEs for network embedding learning. Deep Neural Network for Graph Representations (DNGR) [59] uses a stacked denoising autoencoder [108] to encode and decode the PPMI matrix via multi-layer perceptrons. Concurrently, Structural Deep Network Embedding (SDNE) [60] uses a stacked autoencoder to preserve the node first-order proximity and second-order proximity jointly. SDNE proposes two loss functions on the outputs of the encoder and the outputs of the decoder separately. The first loss function enables the learned network embeddings to preserve the node first-order proximity by minimizing the distance between a node's network embedding and its neighbors' network embeddings. The first loss function L_{1st} is defined as

$$L_{1st} = \sum_{(v,u) \in E} A_{v,u} \|enc(\mathbf{x}_v) - enc(\mathbf{x}_u)\|^2, \quad (29)$$

where $\mathbf{x}_v = \mathbf{A}_{v,:}$ and $enc(\cdot)$ is an encoder which consists of a multi-layer perceptron. The second loss function enables the learned network embeddings to preserve the node second-order proximity by minimizing the distance between a node's inputs and its reconstructed inputs. Concretely, the second loss function L_{2nd} is defined as

$$L_{2nd} = \sum_{v \in V} \|(dec(enc(\mathbf{x}_v)) - \mathbf{x}_v) \odot \mathbf{b}_v\|^2, \quad (30)$$

where $b_{v,u} = 1$ if $A_{v,u} = 0$, $b_{v,u} = \beta > 1$ if $A_{v,u} = 1$, and $dec(\cdot)$ is a decoder which consists of a multi-layer perceptron.

図: 3-3 2nd パラグラフ (Section6-A)

第二パラグラフでは、初期の研究としては MLP(Multi Layer Perceptron) を用いていたことに触れ、また研究例として DNGR(Deep Neural Network for Graph Representation) や SDNE(Structural Deep Network Embedding) について紹介されています。SDNE は学習にあたって二つの誤差関数 (loss) を定義しており、式 (29) で L_{1st} 、式 (30) で L_{2nd} が紹介されています。

DNGR [59] and SDNE [60] only consider node structural information which is about the connectivity between pairs of nodes. They ignore nodes may contain feature information that depicts the attributes of nodes themselves. Graph Autoencoder (GAE*) [61] leverages GCN [22] to encode node structural information and node feature information at the same time. The encoder of GAE* consists of two graph convolutional layers, which takes the form

$$\mathbf{Z} = enc(\mathbf{X}, \mathbf{A}) = Gconv(f(Gconv(\mathbf{A}, \mathbf{X}; \Theta_1)); \Theta_2), \quad (31)$$

where \mathbf{Z} denotes the network embedding matrix of a graph, $f(\cdot)$ is a ReLU activation function and the $Gconv(\cdot)$ function is a graph convolutional layer defined by Equation 12. The decoder of GAE* aims to decode node relational information from their embeddings by reconstructing the graph adjacency matrix, which is defined as

$$\hat{\mathbf{A}}_{v,u} = dec(\mathbf{z}_v, \mathbf{z}_u) = \sigma(\mathbf{z}_v^T \mathbf{z}_u), \quad (32)$$

where \mathbf{z}_v is the embedding of node v . GAE* is trained by minimizing the negative cross entropy given the real adjacency matrix \mathbf{A} and the reconstructed adjacency matrix $\hat{\mathbf{A}}$.

図: 3-4 3rd パラグラフ (Section6-A)

第三パラグラフでは、DNGR や SDNE について触れた後に GAE*についても触れられています。GAE*は二つのグラフ畳み込み層 (graph convolutional layers) で構成されているとされています。以降の内容については省略します。

3.1.2 Graph Generation(Section6-B)

3-1-2 節では Section6-B の Graph Generation について取り扱います。

B. Graph Generation

With multiple graphs, GAEs are able to learn the generative distribution of graphs by encoding graphs into hidden representations and decoding a graph structure given hidden representations. The majority of GAEs for graph generation are designed to solve the molecular graph generation problem, which has a high practical value in drug discovery. These methods either propose a new graph in a sequential manner or in a global manner.

図: 3-5 1st パラグラフ (Section6-B)

第一パラグラフでは話の導入として、GAEs はグラフを隠れ層の表現 (hidden representations) にエンコードし、そこからグラフの構造にデコードすることでグラフの生成分布を学習させるとしています。GAEs を用いたグラフの生成 (graph generation) は創薬 (drug discovery) などにおける分子のグラフ生成の問題を解くことに応用できるとされています。また、アプローチとして sequential manner と global manner の二つが紹介されています。

Sequential approaches generate a graph by proposing nodes and edges step by step. Gomez et al. [111], Kusner et al. [112], and Dai et al. [113] model the generation process of a string representation of molecular graphs named SMILES with deep CNNs and RNNs as the encoder and the decoder respectively.

While these methods are domain-specific, alternative solutions are applicable to general graphs by means of iteratively adding nodes and edges to a growing graph until a certain criterion is satisfied. Deep Generative Model of Graphs (DeepGMG) [65] assumes the probability of a graph is the sum over all possible node permutations:

$$p(G) = \sum_{\pi} p(G, \pi), \quad (37)$$

where π denotes a node ordering. It captures the complex joint probability of all nodes and edges in the graph. DeepGMG generates graphs by making a sequence of decisions, namely whether to add a node, which node to add, whether to add an edge, and which node to connect to the new node. The decision process of generating nodes and edges is conditioned on the node states and the graph state of a growing graph updated by a RecGNN. In another work, GraphRNN [66] proposes a graph-level RNN and an edge-level RNN to model the generation process of nodes and edges. The graph-level RNN adds a new node to a node sequence each time while the edge-level RNN produces a binary sequence indicating connections between the new node and the nodes previously generated in the sequence.

図: 3-6 2nd パラグラフ (Section6-B)

第二パラグラフでは、逐次的アプローチ (Sequential approaches) についてまとめられています。具体的な研究例として DeepGMG (Deep Generative Model of Graphs) について紹介されており、数式 (37) のように順番に node や edge を追加していくことについて記述されています。

Global approaches output a graph all at once. Graph Variational Autoencoder (GraphVAE) [67] models the existence of nodes and edges as independent random variables. By assuming the posterior distribution $q_\phi(\mathbf{z}|G)$ defined by an encoder and the generative distribution $p_\theta(G|\mathbf{z})$ defined by a decoder, GraphVAE optimizes the variational lower bound:

$$L(\phi, \theta; G) = E_{q_\phi(z|G)}[-\log p_\theta(G|\mathbf{z})] + KL[q_\phi(\mathbf{z}|G)||p(\mathbf{z})], \quad (38)$$

where $p(\mathbf{z})$ follows a Gaussian prior, ϕ and θ are learnable parameters. With a ConvGNN as the encoder and a simple multi-layer perception as the decoder, GraphVAE outputs a generated graph with its adjacency matrix, node attributes and edge attributes. It is challenging to control the global properties of generated graphs, such as graph connectivity, validity, and node compatibility. Regularized Graph Variational Au-

(中略)

normalizing a co-occurrence matrix of nodes computed based on random walks produced by the generator.

図: 3-7 3rd パラグラフ (Section6-B)

第三パラグラフでは、グラフを一度に生成する大域的アプローチ (Global approaches) についてまとめられています。具体的な研究例として GraphVAE(Graph Variational Autoencoder) について紹介されており、数式 (38) を最適化してパラメータを求めるところです。

In brief, sequential approaches linearize graphs into sequences. They can lose structural information due to the presence of cycles. Global approaches produce a graph all at once. They are not scalable to large graphs as the output space of a GAE is up to $O(n^2)$.

図: 3-8 4th パラグラフ (Section6-B)

第四パラグラフでは、逐次的アプローチ (Sequential approaches) と大域的アプローチ (Global approaches) についての比較をまとめています。逐次的アプローチではサイクルの存在によって構造的な情報が失われる可能性について、大域的アプローチでは大規模処理に適用しづらいという点について指摘されています。

3.2 STGNNs

3-2 節では Sectio7 の Spatial-Temporal Graph Neural Networks について確認していきます。

VII. SPATIAL-TEMPORAL GRAPH NEURAL NETWORKS

Graphs in many real-world applications are dynamic both in terms of graph structures and graph inputs. Spatial-temporal graph neural networks (STGNNs) occupy important positions in capturing the dynamicity of graphs. Methods under this category aim to model the dynamic node inputs while assuming interdependency between connected nodes. For example, a traffic network consists of speed sensors placed on roads where edge weights are determined by the distance between pairs of sensors. As the traffic condition of one road may depend on its adjacent roads' conditions, it is necessary to consider spatial dependency when performing traffic speed forecasting. As a solution, STGNNs capture spatial and temporal dependencies of a graph simultaneously. The task of STGNNs can be forecasting future node values or labels, or predicting spatial-temporal graph labels. STGNNs follow two directions, RNN-based methods and CNN-based methods.

図: 3-9 1st パラグラフ (Section7)

4
8

第一パラグラフでは、STGNNs(Spatial-Temporal Graph Neural Networks)の背景としてグラフのダイナミズムを取り扱うとしています。ダイナミズムとしてはグラフの構造(graph structure) やグラフの入力(graph inputs) の双方が挙げられています。具体例としては道路に配置された速度センサーから構成される交通のネットワークが挙げられており、交通状況は近隣の道路の状態に依存するため、解決策として空間的かつ時間的なグラフの依存性を同時に STGNNs は考慮する必要があるとされています。STGNNs は二つの研究の方向性があり、RNN-based な手法と CNN-based な手法があると紹介されています。

Most RNN-based approaches capture spatial-temporal dependencies by filtering inputs and hidden states passed to a recurrent unit using graph convolutions [48], [71], [72]. To illustrate this, suppose a simple RNN takes the form

$$\mathbf{H}^{(t)} = \sigma(\mathbf{W}\mathbf{X}^{(t)} + \mathbf{U}\mathbf{H}^{(t-1)} + \mathbf{b}), \quad (39)$$

where $\mathbf{X}^{(t)} \in \mathbf{R}^{n \times d}$ is the node feature matrix at time step t .

After inserting graph convolution, Equation 39 becomes

$$\mathbf{H}^{(t)} = \sigma(Gconv(\mathbf{X}^{(t)}, \mathbf{A}; \mathbf{W}) + Gconv(\mathbf{H}^{(t-1)}, \mathbf{A}; \mathbf{U}) + \mathbf{b}), \quad (40)$$

where $Gconv(\cdot)$ is a graph convolutional layer. Graph Convolutional Recurrent Network (GCRN) [71] combines a LSTM network with ChebNet [21]. Diffusion Convolutional Recurrent Neural Network (DCRNN) [72] incorporates a proposed diffusion graph convolutional layer (Equation 18) into a GRU network. In addition, DCRNN adopts an encoder-decoder framework to predict the future K steps of node values.

図: 3-10 2nd パラグラフ (Section7)

第二パラグラフでは、spatial-temporal な依存性を取得する RNN-based なアプローチについて紹介されています。シンプルな数式として数式 (39) が、数式 (39) をベースに畠み込みを導入した数式として数式 (40) が紹介されています。また研究例として GCRN(Graph Convolutional Recurrent Network) や DCRNN(Diffusion Convolutional Recurrent Neural Network) が紹介されています。

Another parallel work uses node-level RNNs and edge-level RNNs to handle different aspects of temporal information. Structural-RNN [73] proposes a recurrent framework to predict node labels at each time step. It comprises two kinds of RNNs, namely a node-RNN and an edge-RNN. The temporal information of each node and each edge is passed through a node-RNN and an edge-RNN respectively. To incorporate the spatial information, a node-RNN takes the outputs of edge-RNNs as inputs. Since assuming different RNNs for different nodes and edges significantly increases model complexity, it instead splits nodes and edges into semantic groups. Nodes or edges in the same semantic group share the same RNN model, which saves the computational cost.

図: 3-11 3rd パラグラフ (Section7)

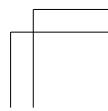
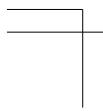
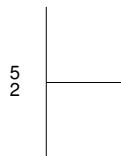
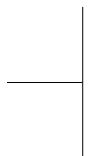
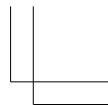
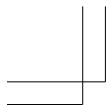
第三パラグラフでは、同様に行われている研究として node-level RNNs や edge-level RNNs が紹介されています。

RNN-based approaches suffer from time-consuming iterative propagation and gradient explosion/vanishing issues. As

alternative solutions, CNN-based approaches tackle spatial-temporal graphs in a non-recursive manner with the advantages of parallel computing, stable gradients, and low memory requirements. As illustrated in Fig 2d, CNN-based approaches interleave 1D-CNN layers with graph convolutional layers to learn temporal and spatial dependencies respectively. Assume the inputs to a spatial-temporal graph neural network is a tensor $\mathcal{X} \in R^{T \times n \times d}$, the 1D-CNN layer slides over $\mathcal{X}_{[:,i,:]}$ along the time axis to aggregate temporal information for each node while the graph convolutional layer operates on $\mathcal{X}_{[i,:,:]}$ to aggregate spatial information at each time step. CGCN [74] integrates 1D convolutional layers with ChebNet [21] or GCN [22] layers. It constructs a spatial-temporal block by stacking a gated 1D convolutional layer, a graph convolutional layer and another gated 1D convolutional layer in a sequential order. ST-GCN [75] composes a spatial-temporal block using a 1D convolutional layer and a PGC layer (Equation 20).

図: 3-12 4th パラグラフ (Section7)

第四パラグラフでは、第二～第三パラグラフで取り扱った RNN-based の欠点として時間がかかる繰り返しの伝播と勾配が爆発したり消失したりの問題について指摘した上で、代替案として CNN-based について紹介しています。CNN-based な手法の利点として、並列処理が可能な点や、勾配が安定している点、メモリ使用が少ない点などについて挙げられています。具体的なアイデアとしては 1D-CNN ベースのアプローチについて紹介されており、具体的な研究例としては CGCN などが挙げられています。



第 4 章

GNNs の実装 (Deep Graph Library)

5
3

第 4 章では、Graph Neural Networks の実装を行うことのできるライブラリである DGL(Deep Graph Library) について取り扱います。4-1 節でインストールについて、4-2 節で簡単な動作確認について、4-3 節では基本事項の確認についてそれぞれ取り扱います。

4.1 インストール

4-1 節ではまずライブラリの DGL(Deep Graph Library) のインストールを行います。

第4章 GNNs の実装 (Deep Graph Library)

4.1 インストール

Install from pip

For CPU builds, run the following command to install with `pip`.

```
pip install dgl
```

For CUDA builds, run one of the following commands and specify the CUDA version.

```
pip install dgl      # For CPU Build
pip install dgl-cu90  # For CUDA 9.0 Build
pip install dgl-cu92  # For CUDA 9.2 Build
pip install dgl-cu100 # For CUDA 10.0 Build
pip install dgl-cu101 # For CUDA 10.1 Build
```

図: 4-1 DGL のインストール (Install from pip)

<https://docs.dgl.ai/install/index.html>

上記に記載がありますが、簡単な動作確認を行うレベルであれば、`pip` を使って CPU 用の DGL をインストールします。

5
4

```
1: $ pip install dgl
```

上記を実行することでインストールを行うことができます。

```
1: $ pip freeze | grep dgl
2: dgl==0.4.2
```

またインストールの確認としては、上記のように "pip freeze | grep dgl" を実行した際に、dgl のバージョンが出力されたら、インストールができています。

4.2 動作確認

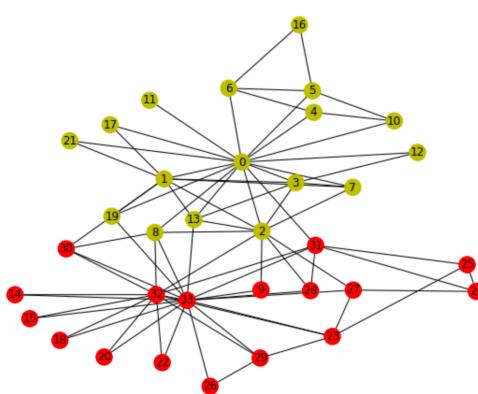
4-2 節では Graph Convolutional Network の学習を行う動作事例の確認を行います。

https://docs.dgl.ai/tutorials/basics/1_first.html

具体的には上記に沿って実行していきます。

Tutorial problem description

The tutorial is based on the "Zachary's karate club" problem. The karate club is a social network that includes 34 members and documents pairwise links between members who interact outside the club. The club later divides into two communities led by the instructor (node 0) and the club president (node 33). The network is visualized as follows with the color indicating the community:



The task is to predict which side (0 or 33) each member tends to join given the social network itself.

図: 4-2 問題設定 (karate クラブのコミュニティ分類)

まず問題設定について確認しますが、上記に記載されています。karate クラブの 34 名のメンバーのソーシャルネットワークを可視化したものであるとされています。クラブのソーシャルネットワークは大きく二つのコミュニ

ティに分けることができるとされており、片方が講師 (instructor; node0) でもう片方がクラブの代表 (club president; node33) であるとされています。上記の図では二つのコミュニティに分けられていますが、タスクはそれぞれのノードが node0 のコミュニティに属するか、node33 のコミュニティに属するかを学習して予測することであるとされています。

Step 1: Creating a graph in DGL

Create the graph for Zachary's karate club as follows:

```
import dgl

def build_karate_club_graph():
    g = dgl.DGLGraph()
    # add 34 nodes into the graph; nodes are labeled from 0~33
    g.add_nodes(34)
    # all 78 edges as a list of tuples
    edge_list = [(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2),
                 (4, 0), (5, 0), (6, 0), (6, 4), (6, 5), (7, 0), (7, 1),
                 (7, 2), (7, 3), (8, 0), (8, 2), (9, 2), (10, 0), (10, 4),
                 (10, 5), (11, 0), (12, 0), (12, 3), (13, 0), (13, 1), (13, 2),
                 (13, 3), (16, 5), (16, 6), (17, 0), (17, 1), (19, 0), (19, 1),
                 (21, 0), (21, 1), (25, 23), (25, 24), (27, 2), (27, 23),
                 (27, 24), (28, 2), (29, 23), (29, 26), (30, 1), (30, 8),
                 (31, 0), (31, 24), (31, 25), (31, 28), (32, 2), (32, 8),
                 (32, 14), (32, 15), (32, 18), (32, 20), (32, 22), (32, 23),
                 (32, 29), (32, 30), (32, 31), (33, 8), (33, 9), (33, 13),
                 (33, 14), (33, 15), (33, 18), (33, 19), (33, 20), (33, 22),
                 (33, 23), (33, 26), (33, 27), (33, 28), (33, 29), (33, 30),
                 (33, 31), (33, 32)]
    # add edges two lists of nodes: src and dst
    src, dst = tuple(zip(*edge_list))
    g.add_edges(src, dst)
    # edges are directional in DGL; make them bi-directional
    g.add_edges(dst, src)

    return g
```

Print out the number of nodes and edges in our newly constructed graph:

```
G = build_karate_club_graph()
print('We have %d nodes.' % G.number_of_nodes())
print('We have %d edges.' % G.number_of_edges())
```

図: Creating a graph in DGL(Step1)

Step1 としてはグラフの作成 (Creating Graph) について記載されています。ここではグラフにノードとエッジを追加することで問題設定にある karate クラブの 34 名のソーシャルネットワークを作成しています。

Out:

```
We have 34 nodes.  
We have 156 edges.
```

図: 4-3 実行結果① (Step1)

この際にグラフの属性を出力したのが上記ですが、78個のエッジを追加しているのにも関わらず156エッジとなっているのは、無向グラフ(undirected graph)においては $0 \rightarrow 1$ と $1 \rightarrow 0$ を同一に見なすためです。隣接行列が常に対称となることからもわかる通り、1つの向きのないエッジの追加は2つの向きのあるエッジの追加と対応していると考えることができます。

```
# Since the actual graph is undirected, we convert it for visualization
# purpose.
nx_G = G.to_networkx().to_undirected()
# Kamada-Kawai layout usually looks pretty for arbitrary graphs
pos = nx.kamada_kawai_layout(nx_G)
nx.draw(nx_G, pos, with_labels=True, node_color=[.7, .7, .7])
```

Figure 1

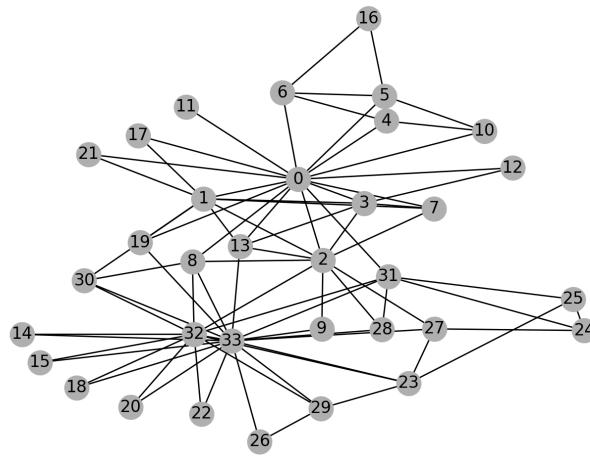


図: 4-4 実行結果② (Step1)

また、可視化を実行すると上記のようになります。一番最初の問題設定の図と同様になっていますが、色がついていないことがポイントです。

Step 2: Assign features to nodes or edges

Graph neural networks associate features with nodes and edges for training. For our classification example, we assign each node an input feature as a one-hot vector: node v_i 's feature vector is $[0, \dots, 1, \dots, 0]$, where the i^{th} position is one.

In DGL, you can add features for all nodes at once, using a feature tensor that batches node features along the first dimension. The code below adds the one-hot feature for all nodes:

図: 4-5 Assign features to nodes or edges(Step2)

Step2 では、ノードやエッジに特徴量 (features) の付与を行なっています。

第4章 GNNs の実装 (Deep Graph Library)

4.2 動作確認

ここでは one-hot ベクトルの形式で特徴量を与えているとされています。

図: 4-6 実行結果③ (Step2)

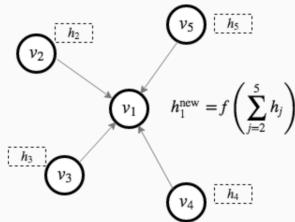
特徴量の追加について記載のコードを実行すると上記のようになります。
上記では、それぞれのノードやエッジに対応するインデックスの位置の数字
が 1 で、他が 0 になっていることが確認できます。

Step 3: Define a Graph Convolutional Network (GCN)

To perform node classification, use the Graph Convolutional Network (GCN) developed by [Kipf and Welling](#). Here is the simplest definition of a GCN framework. We recommend that you read the original paper for more details.

- At layer l , each node v_i^l carries a feature vector h_i^l .
- Each layer of the GCN tries to aggregate the features from u_i^l where u_i^l 's are neighborhood nodes to v into the next layer representation at v_i^{l+1} . This is followed by an affine transformation with some non-linearity.

The above definition of GCN fits into a message-passing paradigm: Each node will update its own feature with information sent from neighboring nodes. A graphical demonstration is displayed below.



Now, we show that the GCN layer can be easily implemented in DGL.

図: 4-7 Define a Graph Convolutional Network(Step3)

Step3 では GCN(Graph Convolutional Network) の定義について記述されています。

<https://arxiv.org/abs/1609.02907>

詳しくは上記の論文の手法をベースにしたとされています。

Step 4: Data preparation and initialization

We use one-hot vectors to initialize the node features. Since this is a semi-supervised setting, only the instructor (node 0) and the club president (node 33) are assigned labels. The implementation is available as follow.

```
inputs = torch.eye(34)
labeled_nodes = torch.tensor([0, 33]) # only the instructor and the president nodes are labeled
labels = torch.tensor([0, 1]) # their labels are different
```

図: 4-8 Data preparation and initialization(Step4)

Step4 ではデータの用意と初期化を行なっています。

Step 5: Train then visualize

The training loop is exactly the same as other PyTorch models. We (1) create an optimizer, (2) feed the inputs to the model, (3) calculate the loss and (4) use autograd to optimize the model.

図: 4-9 Train then visualize(Step5)

Step5 では学習と可視化について記載されています。基本的には PyTorch のモデルと同様の学習を行うとされています。

```
optimizer = torch.optim.Adam(net.parameters(), lr=0.01)
all_logits = []
for epoch in range(30):
    logits = net(G, inputs)
    # we save the logits for visualization later
    all_logits.append(logits.detach())
    logp = F.log_softmax(logits, 1)
    # we only compute loss for labeled nodes
    loss = F.nll_loss(logp[labeled_nodes], labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    print('Epoch %d | Loss: %.4f' % (epoch, loss.item()))
```

```
Epoch 0 | Loss: 1.5280
Epoch 1 | Loss: 1.1456
Epoch 2 | Loss: 0.8636
Epoch 3 | Loss: 0.6770
Epoch 4 | Loss: 0.5631
Epoch 5 | Loss: 0.4871
Epoch 6 | Loss: 0.4343
Epoch 7 | Loss: 0.3932
Epoch 8 | Loss: 0.3635
Epoch 9 | Loss: 0.3345
Epoch 10 | Loss: 0.3063
Epoch 11 | Loss: 0.2801
Epoch 12 | Loss: 0.2533
Epoch 13 | Loss: 0.2246
Epoch 14 | Loss: 0.1927
Epoch 15 | Loss: 0.1563
```

図: 4-10 実行結果④ (Step5)

コードを元に実行すると上記のように実行ができます。`loss.backward()`で誤差逆伝播を行い、`optimizer.step`で学習を行うということは PyTorch の用法とさほど変わらないと考えておけば良さそうです。

4.3 基本事項

https://docs.dgl.ai/tutorials/basics/2_basics.html

上記を元にグラフの作成やノードやエッジの読み書きについて確認し

ます。

4.3.1 Creating a graphについて

4-2-1 節では Creating a graph の項目について確認していきます。

Creating a graph

The design of `DGLGraph` was influenced by other graph libraries. You can create a graph from `networkx` and convert it into a `DGLGraph` and vice versa.

```
import networkx as nx
import dgl

g_nx = nx.petersen_graph()
g_dgl = dgl.DGLGraph(g_nx)

import matplotlib.pyplot as plt
plt.subplot(121)
nx.draw(g_nx, with_labels=True)
plt.subplot(122)
nx.draw(g_dgl.to_networkx(), with_labels=True)

plt.show()
```

図: 4-11 Creating a graph

上記では「DGLGraph の設計は NetworkX に基づいて行われており、NetworkX の形式から DGLGraph の形式に変換する」と記載されています。

```
import networkx as nx
import dgl

g_nx = nx.petersen_graph()
g_dgl = dgl.DGLGraph(g_nx)

import matplotlib.pyplot as plt
plt.subplot(121)
nx.draw(g_nx, with_labels=True)
plt.subplot(122)
nx.draw(g_dgl.to_networkx(), with_labels=True)

plt.show()
```

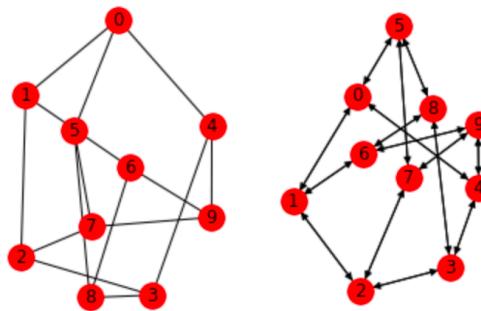


図: 4-12 実行結果①

記載のコードを実行すると上記のようになります。dgl.DGLGraph(g_nx)のところで NetworkX の形式を DGLGraph の形式に変換しています。また、g_dgl.to_networkx() のところで DGLGraph の形式を NetworkX の形式に変換を行なっています。また、それぞれの違いとしては、NetworkX のところでは無向グラフになっていたのが、DGLGraph の形式を経ることで有向グラフの形式に変わっていることが確認できます。NetworkXについては下記のシリーズでもまとめたので、下記も参考にしていただけたらと思います。

<https://lib-arts.hatenablog.com/archive/category/NetworkX>

引き続きチュートリアルを読み進めます。

You can also create a graph by calling the DGL interface.

In the next example, you build a star graph. `DGLGraph` nodes are a consecutive range of integers between 0 and `number_of_nodes()` and can grow by calling `add_nodes`. `DGLGraph` edges are in order of their additions. Note that edges are accessed in much the same way as nodes, with one extra feature: *edge broadcasting*.

```
import dgl
import torch as th

g = dgl.DGLGraph()
g.add_nodes(10)
# A couple edges one-by-one
for i in range(1, 4):
    g.add_edge(i, 0)
# A few more with a paired list
src = list(range(5, 8)); dst = [0]*3
g.add_edges(src, dst)
# finish with a pair of tensors
src = th.tensor([8, 9]); dst = th.tensor([0, 0])
g.add_edges(src, dst)

# Edge broadcasting will do star graph in one go!
g.clear(); g.add_nodes(10)
src = th.tensor(list(range(1, 10)));
g.add_edges(src, 0)

import networkx as nx
import matplotlib.pyplot as plt
nx.draw(g.to_networkx(), with_labels=True)
plt.show()
```

図: 4-13 DGL インターフェースを用いたグラフの作成

上記では、DGL のインターフェースを用いたグラフの作成について記載されています。`add_nodes` でノードの追加、`add_edges` でエッジが追加できると抑えておけば一旦十分かと思います。

```
import dgl
import torch as th

g = dgl.DGLGraph()
g.add_nodes(10)
# A couple edges one-by-one
for i in range(1, 4):
    g.add_edge(i, 0)
# A few more with a paired list
src = list(range(5, 8)); dst = [0]*3
g.add_edges(src, dst)
# finish with a pair of tensors
src = th.tensor([8, 9]); dst = th.tensor([0, 0])
g.add_edges(src, dst)

# Edge broadcasting will do star graph in one go!
g.clear(); g.add_nodes(10)
src = th.tensor(list(range(1, 10)));
g.add_edges(src, 0)

import networkx as nx
import matplotlib.pyplot as plt
nx.draw(g.to_networkx(), with_labels=True)
plt.show()
```

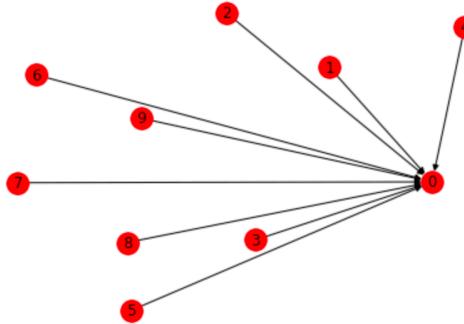


図: 4-14 実行結果②

コードの実行結果は上記のようになります。

4.3.2 Assigning a featureについて

4-2-2 節では Assigning a feature の項目について確認していきます。

Assigning a feature

You can also assign features to nodes and edges of a `DGLGraph`. The features are represented as dictionary of names (strings) and tensors, called **fields**.

The following code snippet assigns each node a vector (`len=3`).

• Note

DGL aims to be framework-agnostic, and currently it supports PyTorch and MXNet tensors. The following examples use PyTorch only.

```
import dgl
import torch as th

x = th.randn(10, 3)
g.ndata['x'] = x
```

`ndata` is a syntax sugar to access the state of all nodes. States are stored in a container `data` that hosts a user-defined dictionary.

```
print(g.ndata['x'] == g.nodes[:,].data['x'])

# Access node set with integer, list, or integer tensor
g.nodes[0].data['x'] = th.zeros(1, 3)
g.nodes[[0, 1, 2]].data['x'] = th.zeros(3, 3)
g.nodes[th.tensor([0, 1, 2])].data['x'] = th.zeros(3, 3)
```

図: 4-15 Assigning a feature

上記では、「DGLGraph 形式のノードやエッジに特徴量をわりあてることができる」と記載されています。また注意書きとして、PyTorch や MXNet の配列 (tensors) の形式を用いており、このチュートリアルでは PyTorch を用いているとされており、コードでも `import torch as th` が記載されています。

```
import dgl
import torch as th

x = th.randn(10, 3)
g.ndata['x'] = x

print(g.ndata['x'] == g.nodes[:].data['x'])

# Access node set with integer, list, or integer tensor
g.nodes[0].data['x'] = th.zeros(1, 3)
g.nodes[[0, 1, 2]].data['x'] = th.zeros(3, 3)
g.nodes[th.tensor([0, 1, 2])].data['x'] = th.zeros(3, 3)

tensor([[True, True, True],
        [True, True, True]])
```

図: 4-16 実行結果③

コードの実行結果は上記のようになっています。基本的に PyTorch の tensor の形式がベースとなっており、様々な配列操作についての確認を行なっています。

第4章 GNNs の実装 (Deep Graph Library)

4.3 基本事項

Assigning edge features is similar to that of node features, except that you can also do it by specifying endpoints of the edges.

```
g.edata['w'] = th.randn(9, 2)

# Access edge set with IDs in integer, list, or integer tensor
g.edges[1].data['w'] = th.randn(1, 2)
g.edges[[0, 1, 2]].data['w'] = th.zeros(3, 2)
g.edges[[th.tensor([0, 1, 2])]].data['w'] = th.zeros(3, 2)

# You can also access the edges by giving endpoints
g.edges[1, 0].data['w'] = th.ones(1, 2) # edge 1 -> 0
g.edges[[1, 2, 3], [0, 0, 0]].data['w'] = th.ones(3, 2) # edges [1, 2, 3] -> 0
```

After assignments, each node or edge field will be associated with a scheme containing the shape and data type (dtype) of its field value.

```
print(g.node_attr_schemes())
g.ndata['x'] = th.zeros((10, 4))
print(g.node_attr_schemes())
```

Out:

```
{'x': Scheme(shape=(3,), dtype=torch.float32)}
{'x': Scheme(shape=(4,), dtype=torch.float32)}
```

You can also remove node or edge states from the graph. This is particularly useful to save memory during inference.

```
g.ndata.pop('x')
g.edata.pop('w')
```

図: 4-17 エッジについての解説

上記ではノードに引き続き、エッジについて説明されています。

```
g.edata['w'] = th.randn(9, 2)

# Access edge set with IDs in integer, list, or integer tensor
g.edges[1].data['w'] = th.randn(1, 2)
g.edges[[0, 1, 2]].data['w'] = th.zeros(3, 2)
g.edges[[th.tensor([0, 1, 2])]].data['w'] = th.zeros(3, 2)

# You can also access the edges by giving endpoints
g.edges[1, 0].data['w'] = th.ones(1, 2)           # edge 1 -> 0
g.edges[[1, 2, 3], [0, 0, 0]].data['w'] = th.ones(3, 2) # edges [1, 2, 3] -> 0

print(g.node_attr_schemes())
g.ndata['x'] = th.zeros(10, 4)
print(g.node_attr_schemes())

g.ndata.pop('x')
g.edata.pop('w')

{'x': Scheme(shape=(3,), dtype=torch.float32)}
{'x': Scheme(shape=(4,), dtype=torch.float32)}

tensor([[ 1.0000,  1.0000],
        [ 1.0000,  1.0000],
        [ 1.0000,  1.0000],
        [ 0.6080,  0.4528],
        [ 0.5203,  0.0516],
        [-0.5324,  0.8116],
        [-0.5677, -0.0175],
        [-1.0647, -1.2786],
        [ 0.7293,  0.0091]])
```

図: 4-18 実行結果④

コードの実行結果は上記のようになります。

本書について

著者プロフィール

下記のブログなどを運営しています。

<https://lib-arts.hatenablog.com/>

お問い合わせなどは Twitter(https://twitter.com/arts_lib) などにお願いいたします。

71

注意事項

- ・著作権は著者に帰属しています。著者に無断での複製は著作権法上での例外を除き禁じます。
- ・本書の内容については著者の力の及ぶ限りでベストを尽くしたものになりますが、誤記などがありましたら気軽にご連絡いただけたら嬉しいです。
- ・本書の内容に関して運用した結果の影響については記載内容に関わらず責任を負いかねますのでご注意ください。

グラフ理論と **Graph Neural Networks** 概論

2020 年 3 月 18 日 初版第 1 刷 発行

著 者 lib-arts
