М. Лабонн, А. Груздев

Графовые нейронные сети на Python

ИЦ «Гевисста»
ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Максим Лабонн, Артем Груздев

Графовые нейронные сети на Python

Maxime Labonne

Hands-On Graph Neural Networks Using Python

BIRMINGHAM—MUMBAI

Максим Лабонн
Артем Груздев

Графовые нейронные сети на Python

ДМК Пресс
издательство
Москва, 2025

ИЦ «Гевисста»

УДК 004.8:004.032.26
ББК 16.6
Л12

Лабонн М., Груздев А.
Л12 Графовые нейронные сети на Python / пер. с англ. А. В. Груздева. – М.: ДМК Пресс, 2024. – 342 с.: ил.

ISBN 978-5-93700-319-5

Графовые нейронные сети стали одной из самых интересных архитектур в глубоком обучении. Технологические компании теперь пытаются применить их повсюду: в системах рекомендаций еды, видео и поиска романтических партнеров, для выявления фейковых новостей, проектирования микросхем и 3D-реконструкции.

Прочитав книгу, вы научитесь создавать собственные графовые наборы из табличных или исходных данных, преобразовывать узлы и ребра в высококачественные эмбеддинги, реализовывать графовые нейронные сети с использованием PyTorch Geometric, выполнять классификацию узлов, генерацию графов, предсказание связей, выбирать лучшую модель в зависимости от вашей задачи.

Издание предназначено специалистам по анализу и обработке данных, а также будет полезно разработчикам на Python и студентам вузов, желающих приобрести знания по одной из самых популярных архитектур ИИ.

УДК 004.8:004.032.26
ББК 16.6

First published in the English language under the title ‘Hands-On Graph Neural Networks Using Python – (9781804617526)’.

ISBN 978-1-80461-752-6 (англ.)
ISBN 978-5-93700-319-5

Содержание

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

ЧАСТЬ I

ВВЕДЕНИЕ В ОБУЧЕНИЕ НА ГРАФАХ

В последние годы представление данных в виде графов становится все более распространенным в различных областях: от социальных сетей до молекулярной биологии. Крайне важно иметь глубокое понимание графовых нейронных сетей (GNN), которые разработаны специально для обработки данных с графовой структурой, чтобы раскрыть весь потенциал этого представления.

Первая часть состоит из двух глав и служит прочной основой для остальной части книги. В ней представлены основные понятия обучения на графах и GNN, а также его актуальность во многих задачах и отраслях. Она также охватывает фундаментальные понятия теории графов, например меры центральности, и их применение в обучении на графах. В этой части также освещаются уникальные особенности и эффективность архитектуры GNN по сравнению с другими методами.

К концу этой части вы получите четкое представление о важности GNN в решении многих реальных проблем. Вы познакомитесь с основами обучения на графах и тем, как оно используется в различных областях. Кроме того, у вас будет полный обзор основных понятий теории графов, которые мы будем использовать в последующих главах. Обладая прочными фундаментальными знаниями, вы сможете перейти к более сложным понятиям, используемым в обучении на графах и GNN в следующих частях книги.

Эта часть состоит из следующих глав:

- глава 1 «Начало работы с обучением на графах»;
- глава 2 «Теория графов для графовых нейронных сетей».

Глава 1

Начало работы с обучением на графах

Добро пожаловать в первую главу нашего путешествия в мир графовых нейронных сетей (GNN). В этой главе мы углубимся в основы GNN и поймем, почему они являются важнейшими инструментами современного анализа данных и машинного обучения. С этой целью мы ответим на три важных вопроса, которые дадут нам полное понимание GNN.

Во-первых, мы изучим важность графов как способа представления данных и почему они широко используются в различных областях, таких как информатика, биология и финансы. Далее углубимся в важность обучения на графах, посмотрим различные примеры применения обучения на графах и различные семейства методов обучения на графах. Наконец, мы сосредоточимся на семействе GNN, подчеркнув его уникальные особенности, производительность и то, чем оно выделяется по сравнению с другими методами.

К концу этой главы у вас будет четкое понимание того, почему GNN важны и как их можно использовать для решения реальных проблем. Вы также получите знания и навыки, необходимые для более глубокого изучения более сложных тем. Итак, начнем!

В этой главе будут рассмотрены следующие основные темы:

- «Почему именно графы?»;
- «Зачем использовать обучение на графах?»;
- «Зачем использовать графовые нейронные сети?».

Почему именно графы?

Первый вопрос, на который нам нужно ответить: почему нас вообще интересуют графы? Теория графов, математическое исследование графов, стала фундаментальным инструментом для понимания сложных систем и отношений. Граф – это визуальное представление набора узлов (также называемых

вершинами) и ребер, которые соединяют эти узлы, обеспечивая структуру для представления сущностей и их отношений (см. рис. 1.1).

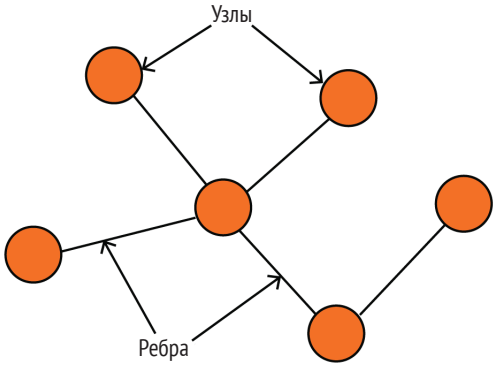


Рис. 1.1 ❖ Пример графа с шестью узлами и пятью ребрами

Представляя сложную систему в виде сети взаимодействующих сущностей, мы можем анализировать их отношения, что позволяет нам глубже понять их основные структуры и закономерности. Универсальность графов делает их популярным выбором в различных областях, включая следующие:

- информатику, где графы можно использовать для моделирования структуры компьютерных программ, это позволяет легче понять, как различные компоненты системы взаимодействуют друг с другом;
- физику: графы можно использовать для моделирования физических систем и их взаимодействий, например взаимосвязей между частицами и их свойствами;
- биологию, где графы можно использовать для моделирования биологической системы, например метаболического пути, в виде сети взаимосвязанных объектов;
- социальные науки, где графы можно использовать для изучения и понимания сложных социальных сетей, включая отношения между людьми в сообществе;
- финансы: графы можно использовать для анализа тенденций фондового рынка и взаимосвязей между различными финансовыми инструментами;
- инженерное дело, где графы можно использовать для моделирования и анализа сложных систем, таких как транспортные сети и электроэнергетические сети.

Эти области знаний естественным образом демонстрируют реляционную структуру. Например, графы являются естественным представлением социальных сетей: узлы – это пользователи, а ребра – дружеские отношения. Но графы настолько универсальны, что их можно применять и к областям, где реляционная структура менее естественна, что открывает новые возможности для открытий и более глубокого анализа.

Например, изображение можно представить в виде графа, как показано на рис. 1.2. Каждый пиксель является узлом, а ребра представляют собой отношения между соседними пикселями. Это позволяет применять графовые алгоритмы к задачам обработки изображений и компьютерного зрения.

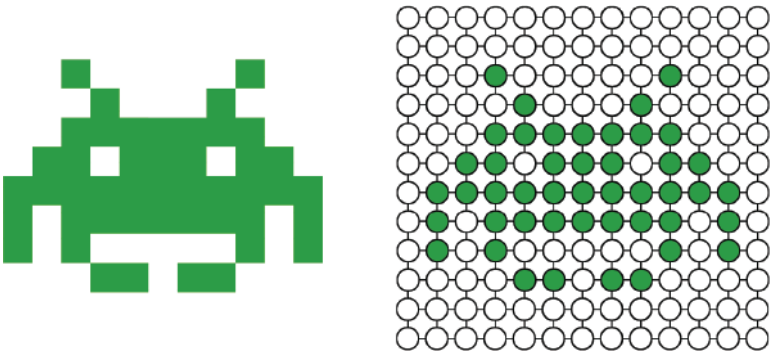


Рис. 1.2 ❖ Слева: исходное изображение; справа: графическое представление этого изображения

Аналогичным образом предложение можно преобразовать в граф, где узлами являются слова, а ребрами – отношения между соседними словами. Этот подход полезен в задачах обработки естественного языка и поиска информации, где контекст и значение слов являются решающими факторами.

В отличие от текста и изображений графы не имеют фиксированной структуры. Однако такая гибкость дополнительно усложняет работу с графами. Отсутствие фиксированной структуры означает, что у них может быть произвольное количество узлов и ребер без определенного порядка. Кроме того, графы могут представлять собой динамические данные, в которых связи между сущностями могут меняться со временем. Например, отношения между пользователями и продуктами могут меняться по мере их взаимодействия друг с другом. В этом сценарии узлы и ребра обновляются, чтобы отражать изменения, происходящие в реальном мире: появление новых пользователей, новых продуктов и возникновение новых отношений.

В следующем разделе мы глубже изучим способы использования графов с помощью машинного обучения с тем, чтобы создать полезные приложения.

Зачем использовать обучение на графах?

Обучение на графах – это применение методов машинного обучения для графовых данных. Эта область исследований охватывает ряд задач, направ-

ленных на анализ и обработку данных с графовой структурой. Существует множество задач обучения на графах, приведем некоторые.

- **Классификация узлов** (node classification) – это задача, которая включает в себя прогнозирование категории (класса) узла в графе. Например, обучение на графах может классифицировать онлайн-пользователей или товары на основе их характеристик. В этой задаче модель обучается на наборе размеченных узлов и их атрибутов и использует эту информацию для прогнозирования класса неразмеченного узла.
- **Прогнозирование появления ребер** (link prediction) – это задача, которая включает в себя прогнозирование отсутствующих ребер между парами узлов графа. Это полезно при заполнении графа знаний, целью которого является составление графа сущностей и их отношений. Например, его можно использовать для прогнозирования отношений между людьми на основе их связей в социальных сетях (рекомендации друзей).
- **Классификация графов** (graph classification) – это задача, которая включает в себя категоризацию различных графов по заранее определенным категориям. Одним из примеров этого является молекулярная биология, в которой молекулярные структуры можно представить в виде графов, и цель состоит в том, чтобы предсказать их свойства для разработки лекарств. В этой задаче модель обучается на наборе размеченных графов и их атрибутов и использует эту информацию для категоризации новых графов.
- **Генерация графов** (graph generation) – это задача, которая включает в себя создание новых графов на основе набора желаемых свойств. Одним из основных приложений является создание новых молекулярных структур для открытия лекарств. Это достигается путем обучения модели на наборе существующих молекулярных структур и последующего ее использования для создания новых, еще неизвестных структур. Сгенерированные структуры можно оценить на предмет их потенциала в качестве кандидатов в лекарства и дополнительно изучить их.

Обучение на графах имеет множество других практических применений, которые могут оказать существенное влияние. Одним из наиболее известных примеров применения **являются рекомендательные системы**, в которых алгоритмы обучения на графах рекомендуют пользователям подходящие товары на основе их предыдущих взаимодействий и отношений с другими элементами. Еще одним важным примером применения является прогнозирование **дорожного движения**, где обучение на графах может улучшить прогнозирование времени в пути за счет учета сложных взаимосвязей между различными маршрутами и видами транспорта.

Универсальность и потенциал обучения на графах делают его интересной областью для исследований и разработки. В последние годы изучение графов быстро продвинулось вперед благодаря наличию больших наборов данных, мощных вычислительных ресурсов, а также достижениям в области

машинного обучения и искусственного интеллекта. В результате мы можем перечислить четыре известных семейства методов обучения на графах [1]:

- **графовая обработка сигналов** (graph signal processing), при которой к графам применяются традиционные методы обработки сигналов, такие как графовое преобразование Фурье и спектральный анализ. Эти методы раскрывают внутренние свойства графа, такие как его связность и структура;
- **факторизация матриц** (matrix factorization), целью которой является поиск низкоразмерных представлений больших матриц. Цель факторизации матрицы – выявить скрытые факторы или закономерности, которые объясняют наблюдаемые отношения в исходной матрице. Этот подход может обеспечить компактное и интерпретируемое представление данных;
- **случайное блуждание** (random walk) – математическая концепция, используемая для моделирования движения сущностей внутри графа. Моделируя случайные блуждания по графу, мы можем собрать информацию о связях между узлами. Вот почему их часто используют для создания обучающих данных для моделей машинного обучения;
- **глубокое обучение** (deep learning) – подобласть машинного обучения, в которой основное внимание уделяется нейронным сетям с несколькими слоями. Методы глубокого обучения могут эффективно кодировать и представлять графовые данные в виде векторов. Эти векторы затем можно будет использовать в различных задачах с замечательной производительностью.

Важно отметить, что эти методы не являются взаимоисключающими и часто пересекаются при решении задач. На практике их часто объединяют, образуя гибридные модели, в которых используются сильные стороны каждой. Например, матричная факторизация и методы глубокого обучения могут использоваться совместно для изучения низкоразмерных представлений данных с графовой структурой.

По мере того как мы углубляемся в мир обучения на графах, крайне важно ясно представлять фундаментальный строительный блок любого метода машинного обучения – набор данных. Традиционные наборы табличных данных, такие как электронные таблицы, представляют данные в виде строк и столбцов, где каждая строка представляет одну точку данных. Однако во многих реальных сценариях отношения между точками данных столь же значимы, как и сами точки данных. Именно здесь на помощь приходят графовые наборы данных (наборы данных в виде графа). Графовые наборы данных графа представляют точки данных в виде узлов графа, а связи между этими точками данных – в виде ребер.

В качестве примера возьмем табличный набор данных, показанный на рис. 1.3.

Этот набор данных содержит информацию о пяти членах семьи. У каждого участника есть три характеристики (или атрибута): имя, возраст и пол.

Однако табличная версия этого набора данных не показывает связи между этими людьми. Напротив, графовая версия представляет их в виде ребер, что позволяет нам понять связи в этом семействе. Во многих контекстах соединения между узлами имеют решающее значение для понимания данных, поэтому представление данных в виде графа становится все более популярным.

Табличный набор данных

ID	Name	Age	Gender
1	Mary	76	Female
2	John	75	Male
3	Kate	46	Female
4	Robert	47	Male
5	Loise	18	Female

Графовый набор данных

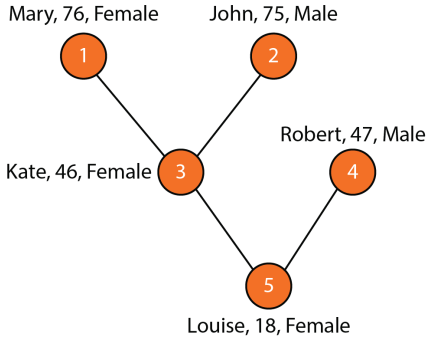


Рис. 1.3 ❖ Генеалогическое древо, представленное в виде табличного набора данных (слева) и графового набора данных (справа)

Теперь, когда у нас есть базовое понимание машинного обучения на графах и различных типов задач, которые оно включает в себя, можем перейти к изучению одного из наиболее важных подходов к решению этих задач: графовых нейронных сетей.

Зачем использовать графовые нейронные сети?

В этой книге мы сосредоточимся на семействе методов глубокого обучения на графах, часто называемых **графовыми нейронными сетями** (graph neural networks – GNN). GNN – это новый тип архитектуры глубокого обучения, специально разработанный для данных, представленных в виде графов. В отличие от традиционных алгоритмов глубокого обучения, которые в первую очередь разрабатывались для текста и изображений, GNN созданы специально для обработки и анализа графовых наборов данных (см. рис. 1.4).

GNN стали мощным инструментом обучения на графах и показали отличные результаты в различных задачах и отраслях. Одним из наиболее ярких примеров является открытие нового антибиотика с помощью GNN [2]. Модель была обучена на 2500 молекулах и протестирована на библиотеке из

6000 соединений. Они предсказали, что молекула под названием халицин сможет убивать многие устойчивые к антибиотикам бактерии, обладая при этом низкой токсичностью для клеток человека. Основываясь на этом прогнозе, исследователи использовали халицин для лечения мышей, инфицированных бактериями, устойчивыми к антибиотикам. Они продемонстрировали ее эффективность и полагают, что эту модель можно использовать для разработки новых лекарств.

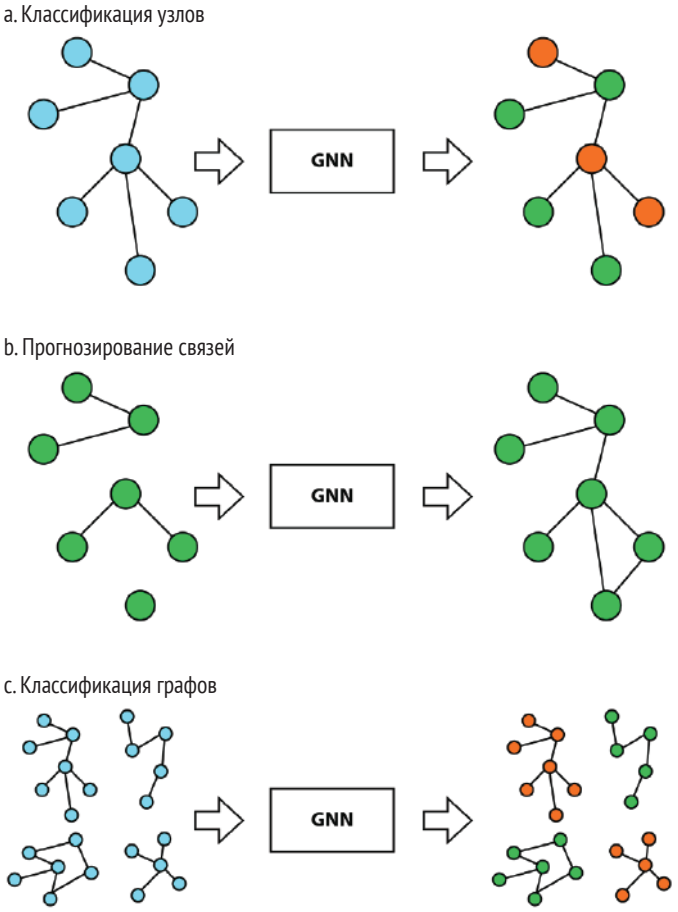


Рис. 1.4 ❖ Высокоуровневая архитектура конвейера GNN с графом на входе и выходом, соответствующим заданной задаче

Как работают GNN? Давайте возьмем пример задачи классификации узлов в социальной сети, аналогичный генеалогическому древу, приведенному ранее на рис. 1.3. В задаче классификации узлов GNN используют информацию из разных источников для создания векторного представления каждого узла в графе. Это представление включает в себя не только исходные признаки узлов (например, имя, возраст и пол), но и информацию о признаках ребер

(например, силу связей между узлами) и глобальные признаки (например, общесетевые статистики).

Вот почему GNN более эффективны, чем традиционные методы машинного обучения на графах. Вместо того чтобы ограничиться исходными атрибутами, GNN обогащают признаки исходного узла атрибутами соседних узлов, ребер и глобальными признаками, что делает представление гораздо более полным и содержательным. Новые представления узлов затем используются для выполнения конкретной задачи, например классификации узлов, регрессии или прогнозирования связей.

В частности, GNN определяют операцию свертки на графе, которая агрегирует информацию из соседних узлов и ребер для обновления представления узлов (представления вершин). Эта операция выполняется итеративно, позволяя модели изучать более сложные взаимосвязи между узлами по мере увеличения количества итераций. Например, на рис. 1.5 показано, как GNN получает представление узла 5, используя соседние узлы.

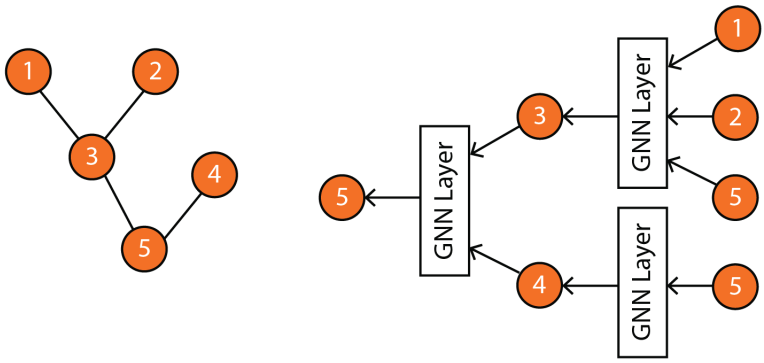


Рис. 1.5 ❖ Слева: граф на входе; справа: вычислительный граф, показывающий, как GNN получает представление узла 5 на основе информации о его соседях

Стоит отметить, что на рис. 1.5 представлена упрощенная иллюстрация вычислительного графа. В действительности существуют различные виды GNN и слоев GNN, каждый из которых имеет уникальную структуру и способ агрегирования информации из соседних узлов. Эти различные варианты GNN также имеют свои преимущества и ограничения и хорошо подходят для конкретных типов графовых данных и задач. При выборе подходящей архитектуры GNN для конкретной задачи крайне важно понимать характеристики графовых данных и желаемый результат.

В более общем плане GNN, как и другие методы глубокого обучения, наиболее эффективны в применении к конкретным задачам. Эти задачи характеризуются высокой сложностью, а это означает, что изучение эффективных представлений имеет важнейшее значение для решения поставленной задачи. Например, очень сложной задачей может быть рекомендация нужных продуктов среди миллиардов имеющихся товаров миллионам клиентов.

С другой стороны, некоторые задачи, такие как поиск самого младшего члена нашего генеалогического древа, можно решить без какого-либо метода машинного обучения.

Более того, для эффективной работы GNN требуется значительный объем данных. Традиционные методы машинного обучения могут лучше подойти в случаях, когда набор данных является небольшим, поскольку они в меньшей степени зависят от больших объемов данных. Однако эти методы не масштабируются так же хорошо, как GNN. GNN могут обрабатывать наборы данных большего размера благодаря параллельному и распределенному обучению. Кроме того, они могут более эффективно использовать дополнительную информацию, а это в свою очередь дает лучшие результаты.

Выводы

В этой главе мы ответили на три основных вопроса: почему именно графы, зачем использовать обучение на графах и зачем использовать графовые нейронные сети? Во-первых, мы исследовали универсальность графов для представления различных типов данных, таких как социальные сети и транспортные сети, а также для представления текста и изображений. Мы обсудили различные примеры применения обучения на графах, включая классификацию узлов и классификацию графов, и выделили четыре основных семейства методов обучения на графах. Наконец, мы подчеркнули важность GNN и их превосходство над другими методами, особенно в отношении больших и сложных наборов данных. Отвечая на эти три основных вопроса, мы стремились дать всесторонний обзор важности GNN и объяснить, почему они становятся жизненно важными инструментами машинного обучения.

В главе 2 «Теория графов для графовых нейронных сетей» мы углубимся в основы теории графов, которая дает основы для понимания GNN. В этой главе будут рассмотрены фундаментальные понятия теории графов, включая такие понятия, как матрицы смежности и степени. Кроме того, мы углубимся в различные типы графов и примеры их применения, такие как ориентированные и неориентированные графы, а также взвешенные и невзвешенные графы.

Дополнительное чтение

[1] F. Xia et al., Graph Learning: A Survey, IEEE Transactions on Artificial Intelligence, vol. 2, no. 2, pp. 109–127, Apr. 2021, DOI: 10.1109/tai.2021.3076021. Доступ по ссылке https://arxiv.org/abs/2105.00696.

[2] A. Trafton, Artificial intelligence yields new antibiotic, MIT News, 20-Feb-2020. [Online]. Доступ по ссылке https://news.mit.edu/2020/artificial-intelligence-identifies-new-antibiotic-0220.

Глава 2

Теория графов для графовых нейронных сетей

Теория графов – фундаментальный раздел математики, занимающийся изучением графов и сетей. Граф – это визуальное представление сложных структур данных, которое помогает нам понять отношения между различными объектами. Теория графов дает нам инструменты для моделирования и анализа широкого спектра реальных задач, в которых объектами внимания могут быть транспортные системы, социальные сети и интернет-соединения.

В этой главе мы окунемся в основы теории графов, охватив три основные темы: свойства графов, понятия теории графов и алгоритмы графов. Начнем с определения графов и их компонент. Затем мы познакомим вас с различными типами графов и раскроем их свойства и области применения. Далее рассмотрим фундаментальные понятия графов, объекты и меры, включая матрицу смежности. Наконец, мы погрузимся в графовые алгоритмы, сосредоточив внимание на двух фундаментальных алгоритмах обхода графа: **поиске в ширину** (breadth-first search – BFS) и **поиске в глубину** (depth-first search – DFS).

К концу этой главы вы получите прочные фундаментальные знания в области теории графов, которая позволит вам решать более сложные темы и проектировать графовые нейронные сети.

В этой главе будут рассмотрены следующие основные темы:

- «Знакомство со свойствами графа»,
- «Знакомство с понятиями теории графов»,
- «Изучение графовых алгоритмов».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter02.

Знакомство со свойствами графа

В теории графов граф – это математическая структура, состоящая из набора объектов, называемых **вершинами** (vertices) или узлами (nodes), и набора соединений, называемых **ребрами** (edges), которые связывают пары вершин. Для представления графа можно использовать обозначение $G = (V, E)$, где G – это граф, V – это набор вершин и E – набор ребер.

Узлы графа могут представлять любые объекты, например города, людей, веб-страницы или молекулы, а ребра представляют отношения или связи между ними, например обычные дороги, социальные отношения, гиперссылки или химические связи.

В этом разделе будет представлен обзор фундаментальных свойств графов, которые будут широко использоваться в последующих главах.

Ориентированные графы (directed graphs)

Прежде всего графы бывают ориентированными и неориентированными. В **ориентированном графе** (directed graph), также называемом **орграфом** (digraph), каждому ребру присвоено направление или ориентация. Это означает, что ребро соединяет два узла в определенном направлении, где один узел является источником, а другой – пунктом назначения. У **неориентированного графа** (undirected graph) неориентированные ребра, т. е. ребрам не присвоено направление. Это означает, что ребро между двумя вершинами можно пройти в любом направлении, и порядок посещения узлов не имеет значения.

В Python мы можем создать неориентированный граф с помощью класса `nx.Graph` библиотеки networkx:

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()
G.add_edges_from(
    [('A', 'B'), ('A', 'C'), ('B', 'D'),
     ('B', 'E'), ('C', 'F'), ('C', 'G')]
)
```

```
plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```

На рис. 2.1 изображен построенный нами неориентированный граф G.

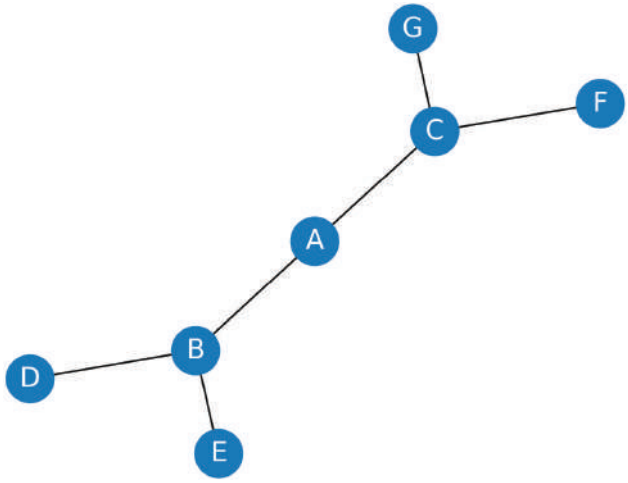


Рис. 2.1 ❖ Пример неориентированного графа

Программный код для построения ориентированного графа будет аналогичным, мы просто заменим класс nx.Graph на класс nx.DiGraph:

```
DG = nx.DiGraph()
DG.add_edges_from(
    [('A', 'B'), ('A', 'C'), ('B', 'D'),
     ('B', 'E'), ('C', 'F'), ('C', 'G')]
)

plt.axis('off')
nx.draw_networkx(DG,
                 pos=nx.spring_layout(G, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```

На рис. 2.2 изображен построенный нами ориентированный граф DG.

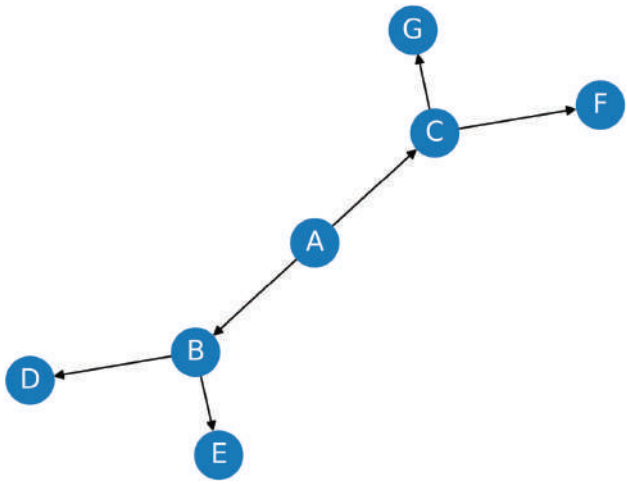


Рис. 2.2 ❖ Пример ориентированного графа

В ориентированных графах ребра обычно обозначаются стрелками, обозначающими их ориентацию, как показано на рис. 2.2.

Взвешенные графы (weighted graphs)

Ребра графа могут быть взвешенными или невзвешенными. Во **взвешенном графе** (weighted graphs) каждое ребро имеет связанный с ним вес или стоимость. Эти веса могут отражать различные факторы, такие как расстояние, время в пути или стоимость.

Например, в транспортной сети веса ребер могут представлять собой расстояния между разными городами или время, необходимое для перемещения между ними. Напротив, невзвешенные графы не имеют весов, связанных с их ребрами. Эти типы графов обычно используются в ситуациях, когда отношения между узлами являются бинарными, а ребра просто указывают на наличие или отсутствие связи между ними.

Мы можем изменить предыдущий неориентированный граф, присвоив веса нашим ребрам. В networkx ребра графа задаются с помощью кортежа, содержащего начальный и конечный узлы, а также словаря, задающего вес ребра:

```
WG = nx.Graph()
WG.add_edges_from(
    [('A', 'B', {"weight": 10}),
     ('A', 'C', {"weight": 20}),
     ('B', 'D', {"weight": 30}),
     ('B', 'E', {"weight": 40}),
     ('C', 'F', {"weight": 50}),
     ('C', 'G', {"weight": 60})]
)
```

```
labels = nx.get_edge_attributes(WG, "weight")

plt.axis('off')
nx.draw_networkx(WG,
                 pos=nx.spring_layout(G, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
nx.draw_networkx_edge_labels(WG,
                             pos=nx.spring_layout(G, seed=0),
                             edge_labels=labels);
```

На рис. 2.3 изображен построенный нами взвешенный граф WG.

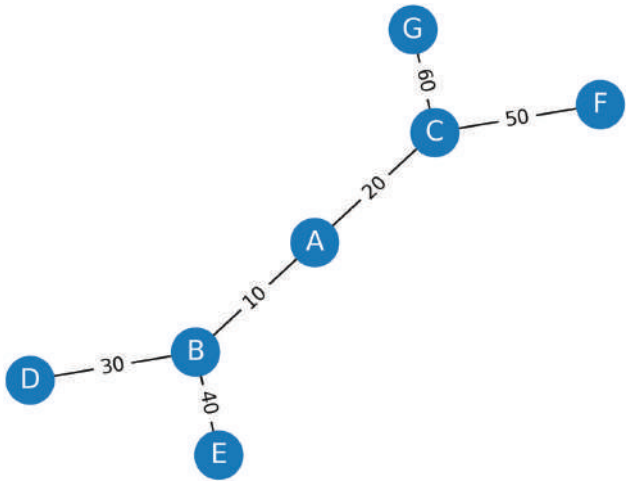


Рис. 2.3 ❖ Пример взвешенного графа

Связные графы (connected graphs)

Связность графов – это фундаментальное понятие теории графов, тесно сопряженное со структурой и функциями графа.

В связном графе между любыми двумя вершинами графа существует путь. Формально граф *G* является связным тогда и только тогда, когда для каждой пары вершин *u* и *v* существует путь из *u* в *v*. Напротив, граф является несвязным, если он не является связным, а это означает, что по крайней мере две его вершины не соединены путем.

Библиотека networkx предлагает встроенную функцию, позволяющую проверить, является граф связным или нет. В следующем примере первый граф, в отличие от второго, содержит изолированные узлы (4 и 5) и поэтому не является связным:

```
G1 = nx.Graph()
G1.add_edges_from([(1, 2), (2, 3), (3, 1), (4, 5)])
print(f"Граф 1 является связным? {nx.is_connected(G1)}")

G2 = nx.Graph()
G2.add_edges_from([(1, 2), (2, 3), (3, 1), (1, 4)])
print(f"Граф 2 является связным? {nx.is_connected(G2)}")
```

```
Граф 1 является связным? False
Граф 2 является связным? True
```

Сами графы показаны на рис. 2.4.

```
plt.figure(figsize=(8,8))

plt.subplot(221)
plt.axis('off')
nx.draw_networkx(G1,
                 pos=nx.spring_layout(G1, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )

plt.subplot(222)
plt.axis('off')
nx.draw_networkx(G2,
                 pos=nx.spring_layout(G2, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```

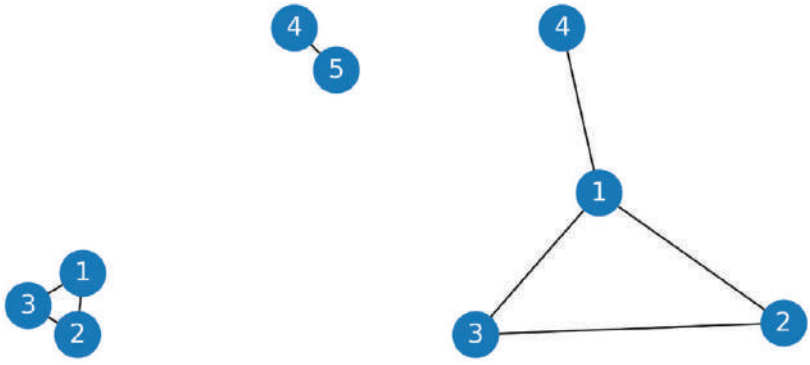


Рис. 2.4 ❖ Слева: граф 1 с изолированными узлами (несвязный граф); справа: граф 2, где каждый узел соединен как минимум с одним другим (связный граф)

У связных графов есть несколько интересных свойств и примеров применения. Например, в коммуникационной сети связный граф гарантирует,

что любые два узла могут взаимодействовать друг с другом с помощью пути. Напротив, у несвязных графов могут быть изолированные узлы, которые не могут взаимодействовать с другими узлами в сети, что затрудняет разработку эффективных алгоритмов маршрутизации.

Существуют разные способы измерения связности графа. Одной из наиболее распространенных мер связности является минимальное количество ребер, которые необходимо удалить для рассоединения графа, которое известно как минимальный разрез графа (graph's minimum cut). Задача минимального разреза имеет ряд применений – в оптимизации сетевых потоков, кластеризации и обнаружении сообществ.

Типы графов

Помимо широко используемых типов графов, существуют некоторые специальные типы графов, обладающие уникальными свойствами и характеристиками.

- **Дерево** (tree) – это связный неориентированный граф без циклов (как граф на рис. 2.1). Поскольку между любыми двумя узлами дерева существует только один путь, дерево является частным случаем графа. Деревья часто используются для моделирования иерархических структур, таких как генеалогические деревья, организационные структуры или деревья классификации.
- **Корневое дерево** (rooted tree) – это дерево, в котором одна вершина помечена как корень, а все остальные вершины соединены с ним уникальным путем. Корневые деревья часто используются в информатике для представления иерархических структур данных, таких как файловые системы или структура XML-документов.
- **Ориентированный ациклический граф** (directed acyclic graph – DAG) – это ориентированный граф, не имеющий циклов (как граф на рис. 2.2). Это означает, что проходить по ребрам можно только в определенном направлении, петель и циклов нет. DAG часто используются для моделирования зависимостей между задачами или событиями – например в управлении проектами или при вычислении критического пути выполнения задания.
- **Двудольный граф** (bipartite graph) – это граф, вершины которого можно разделить на два непересекающихся множества, так что все ребра соединяют вершины в разных множествах. Двудольные графы часто используются в математике и информатике для моделирования отношений между двумя разными типами объектов, такими как покупатели и продавцы или сотрудники и проекты.
- **Полный граф** (complete graph) – это граф, в котором каждая пара вершин соединена ребром. Полные графы часто используются в комбинаторике для моделирования задач, включающих все возможные по-

парные соединения, а также в компьютерных нейронных сетях для моделирования полносвязных нейронных сетей.

Рисунок 2.5 иллюстрирует разные типы графов.

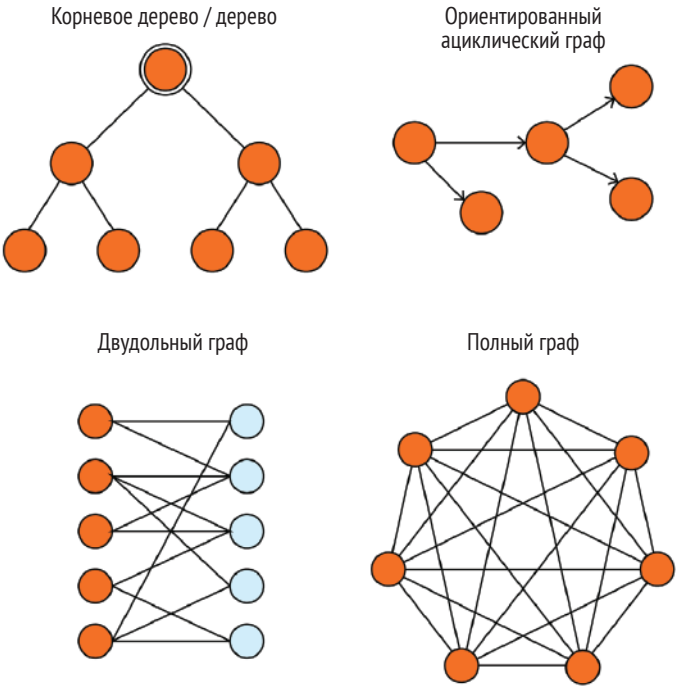


Рис. 2.5 ❖ Распространенные типы графов

Теперь, когда мы рассмотрели основные типы графов, давайте перейдем к изучению некоторых наиболее важных понятий теории графов. Знакомство с этими понятиями поможет нам эффективно анализировать графы и работать с ними.

Знакомство с основными понятиями теории графов

В этом разделе мы рассмотрим некоторые основные понятия теории графов, включая объекты графа (такие как степень и соседи), меры графа (такие как центральность и плотность) и представление графа в виде матрицы смежности.

Фундаментальные объекты

Одним из ключевых понятий теории графов является **степень** (degree) узла, которая представляет собой количество ребер, **инцидентных** (incident) этому узлу. Говорят, что ребро инцидентно узлу, если этот узел является одной из конечных точек этого ребра. Степень узла v часто обозначается $\deg(v)$. Ее можно определить как для ориентированных, так и для неориентированных графов:

- в неориентированном графе степень вершины – это количество ребер, соединенных с вершиной. Обратите внимание, что, если узел соединен сам с собой (так называемая **петля** (loop) или **самопетля** (self-loop)), он добавляет 2 в степень вершины;
- в ориентированном графе степень делится на два типа: **входящую** (indegree) и **исходящую** (outdegree). Входящая степень узла (обозначается как $\deg^-(v)$) – это количество ребер, которые входят в этот узел, а исходящая степень узла (обозначается $\deg^+(v)$) – это количество ребер, которые выходят из этого узла. В данном случае самопетля добавляет единицу во входящую и исходящую степени.

Входящая и исходящая степени необходимы для анализа и понимания ориентированных графов, поскольку они дают представление о том, как информация или ресурсы распределяются внутри графа. Например, узлы с высокой входящей степенью, вероятно, будут важными источниками информации или ресурсов. Напротив, узлы с высокой исходящей степенью, скорее всего, будут важными пунктами назначения или важными потребителями информации или ресурсов.

В networkx мы можем довольно просто вычислить степень узла, входящую или исходящую степень, используя встроенные методы. Давайте сделаем это для неориентированного графа с рис. 2.1 и ориентированного графа с рис. 2.2:

```
G = nx.Graph()
G.add_edges_from(
    [('A', 'B'), ('A', 'C'), ('B', 'D'),
     ('B', 'E'), ('C', 'F'), ('C', 'G')]
)
print(f"deg(A) = {G.degree['A']}")

DG = nx.DiGraph()
DG.add_edges_from(
    [('A', 'B'), ('A', 'C'), ('B', 'D'),
     ('B', 'E'), ('C', 'F'), ('C', 'G')]
)
print(f"deg^-(A) = {DG.in_degree['A']}")
print(f"deg^+(A) = {DG.out_degree['A']}")
```

```
deg(A) = 2
deg^-(A) = 0
deg^+(A) = 2
```

Для неориентированного графа степень узла A равна 2: $\deg(A) = 2$, поскольку только два узла соединены с ним. Теперь смотрим ориентированный граф. Узел A соединен с двумя ребрами, поэтому исходящая степень равна 2 ($\deg(A) = \deg^{+}(A) = 2$), но не является пунктом назначения ни для одного из них, поэтому входящая степень равна 0 ($\deg^{-}(A) = 0$).

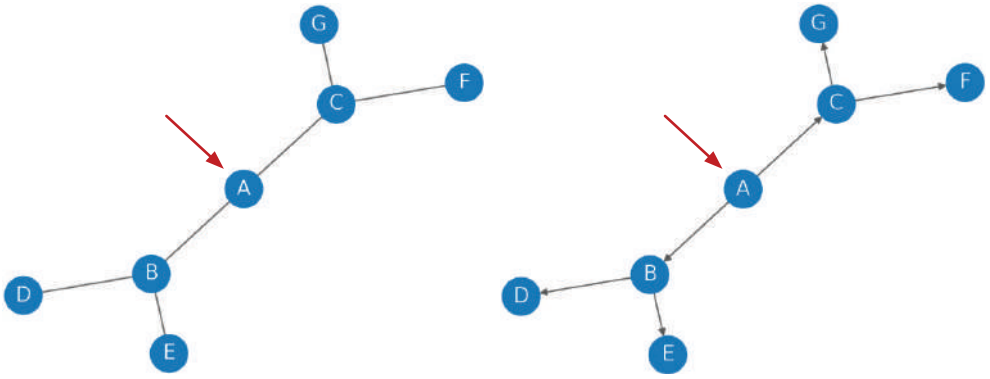


Рис. 2.6 ❖ Сравнение неориентированного графа (слева) и ориентированного графа (справа) по степени узла A, входящей и исходящей степеней узла A

Понятие степени узла связано с понятием **соседей** (neighbors). Соседи – это узлы, напрямую соединенные с конкретным узлом с помощью ребра. Более того, два узла называются **смежными** (adjacent), если у них есть хотя бы один общий сосед. Понятия соседей и смежности являются фундаментальными для многих графовых алгоритмов и примеров применения, таких как поиск **пути** (path) между двумя узлами или идентификация кластеров в сети.

В теории графов путь – это последовательность ребер, соединяющих два (или более) узла графа. Длина пути – это количество ребер, пройденных по пути. Существуют разные типы путей, но два из них особенно важны:

- **простой путь** (simple path) – это путь, в котором ни один узел не посещается дважды, за исключением начальной и конечной вершин;
- **цикл** (cycle) – это путь, у которого первая и последняя вершины совпадают. Граф называется ациклическим, если он не содержит циклов (например, деревья и DAG'и).

Степени и пути можно использовать для определения важности узла в сети. Эта мера важности называется **центральностью** (centrality).

Меры центральности для графов

Центральность количественно определяет важность вершины или узла в сети. Она помогает нам идентифицировать ключевые узлы в графе на основе их связности и влияния на поток информации или взаимодействия

внутри сети. Существует несколько мер центральности, каждая из которых дает различный взгляд на важность узла.

- **Центральность по степени** (degree centrality) – одна из самых простых и наиболее часто используемых мер центральности. Она просто определяется как степень узла. Высокая степень центральности указывает на то, что вершина тесно связана с другими вершинами графа и, таким образом, существенно влияет на сеть.
- **Центральность по близости** (closeness centrality) измеряет, насколько близко узел расположен ко всем остальным узлам графа. Он соответствует средней длине кратчайшего пути между исследуемым узлом и всеми остальными узлами графа. Из вершины с высокой центральностью по близости можно быстро достичь всех остальных вершин сети.
- **Центральность по промежуточности или посредничеству** (betweenness centrality) измеряет, сколько раз данный узел находился на кратчайшем пути *между* любыми двумя другими узлами в графе. Узел с высокой центральностью по промежуточности выполняет роль узкого места или моста между различными частями графа. Такой узел позволяет отслеживать или контролировать поток информации в сети.

Давайте вычислим эти меры для нашего неориентированного графа `G`, используя встроенные функции networkx, и проанализируем результат:

```
print(f"Degree centrality      = {nx.degree_centrality(G)}")
print(f"Closeness centrality   = {nx.closeness_centrality(G)}")
print(f"Betweenness centrality = {nx.betweenness_centrality(G)}")
```

```
Degree centrality      = {'A': 0.3333333333333333, 'B': 0.5, 'C': 0.5,
                          'D': 0.16666666666666666, 'E': 0.16666666666666666,
                          'F': 0.16666666666666666, 'G': 0.16666666666666666}
Closeness centrality   = {'A': 0.6, 'B': 0.5454545454545454, 'C': 0.5454545454545454,
                          'D': 0.375, 'E': 0.375, 'F': 0.375, 'G': 0.375}
Betweenness centrality = {'A': 0.6, 'B': 0.6, 'C': 0.6, 'D': 0.0,
                          'E': 0.0, 'F': 0.0, 'G': 0.0}
```

Мы получаем словари, в которых ключи – это узлы (вершины), а значения – значения центральности.

Важность узлов *A*, *B* и *C* в графе зависит от типа использованной центральности. Центральность по степени считает узлы *B* и *C* более важными, поскольку у них больше соседей, чем у узла *A*. Однако с точки зрения центральности по близости узел *A* является наиболее важным, поскольку из него можно достичь любого другого узла графа по кратчайшему пути. С другой стороны, узлы *A*, *B* и *C* имеют равную центральность по промежуточности, они играют очень важную роль на пути «между» парами других узлов сети в том смысле, что пути между другими узлами должны проходить через эти узлы. Еще можно сказать так: узлы *A*, *B* и *C* входят в большое количество кратчайших путей между другими узлами.

В дополнение к этим мерам в следующих главах мы увидим, как рассчитать важность узла с помощью методов машинного обучения. Однако это не единственная мера, которую мы рассмотрим.

Действительно, **плотность** (density) – еще одна важная мера, измеряющая связность графа. Это отношение фактического количества ребер к максимально возможному количеству ребер в графе. Граф с высокой плотностью считается более связным и характеризуется более интенсивным потоком информации по сравнению с графом низкой плотности.

Формула расчета плотности зависит от того, является ли граф ориентированным или неориентированным. Для неориентированного графа с n узлами максимально возможное количество ребер равно $\frac{n(n-1)}{2}$.

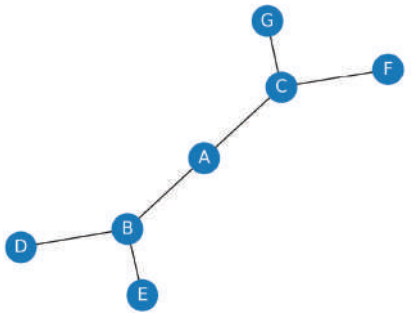
Для ориентированного графа с n узлами максимально возможное количество ребер равно $n(n-1)$.

Например, у графа на рис. 2.1 – 6 ребер и максимально возможное количество ребер $\frac{7(7-1)}{2} = 21$. Следовательно, плотность графа будет равна $\frac{6}{21} \approx 0.2857$.

У плотного графа плотность ближе к 1, у разреженного графа плотность ближе к 0. Не существует строгого правила относительно того, что представляет собой плотный или разреженный граф, но обычно граф считается плотным, если его плотность больше 0.5, и разреженным, если его плотность меньше 0.1. Эта мера напрямую связана с фундаментальной проблемой графов – представлением графа в виде матрицы смежности.

Представление графа в виде матрицы смежности

Матрица смежности – это матрица, в которой хранится информация о ребрах графа. В ней каждая ячейка указывает на отсутствие или наличие ребра между двумя узлами. Матрица смежности является квадратной матрицей размером $n \times n$, где n – это количество узлов в графе. Значение 1 в ячейке (i, j) указывает на наличие ребра между узлом i и узлом j, тогда как значение 0 указывает на отсутствие ребра между ними. Для неориентированного графа матрица является симметричной, тогда как для ориентированного графа матрица не обязательно должна быть симметричной. На рис. 2.7 показана матрица смежности для нашего неориентированного графа с рис. 2.1.



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

Рис. 2.7 ❖ Пример матрицы смежности

В Python матрицу смежности можно реализовать в виде списка списков, как показано в этом примере:

```
adj = [[0,1,1,0,0,0,0],
       [1,0,0,1,1,0,0],
       [1,0,0,0,0,1,1],
       [0,1,0,0,0,0,0],
       [0,1,0,0,0,0,0],
       [0,0,1,0,0,0,0],
       [0,0,1,0,0,0,0]]
```

Матрица смежности – это простое представление, которое можно легко визуализировать в виде двумерного массива. Одно из ключевых преимуществ матрицы смежности заключается в том, что проверка соединения между двумя узлами является операцией с постоянным временем. Это позволяет эффективно использовать ее для проверки наличия ребра в графе. Более того, она используется для выполнения матричных операций, которые полезны для определенных алгоритмов на графах, таких как вычисление кратчайшего пути между двумя узлами.

Однако добавление или удаление узлов может оказаться дорогостоящим, поскольку необходимо изменить размер матрицы или сдвинуть матрицу. Одним из основных недостатков использования матрицы смежности является ее пространственная сложность: по мере роста количества узлов в графе пространство, необходимое для хранения матрицы смежности, увеличивается в геометрической прогрессии. Формально мы можем сказать, что матрица смежности имеет пространственную сложность $O(|V|^2)$, где $|V|$ – это количество узлов в графе.

В целом, хотя матрица смежности является полезной структурой данных для представления небольших графов, она может быть непрактична для более крупных графов из-за своей пространственной сложности. Кроме того, вычислительные затраты на добавление или удаление узлов могут сделать его неэффективным для динамически изменяющихся графов.

Вот почему другие представления могут быть тоже полезны. Например, еще один популярный способ хранения графов – **список ребер** (edge list). Список ребер – это список всех ребер графа. Каждое ребро представлено кортежем с парой вершин. Кроме того, список ребер может включать вес или стоимость каждого ребра. Это как раз та структура данных, которую мы использовали для создания наших графов с помощью библиотеки networkx:

```
edge_list = [(0, 1), (0, 2), (1, 3),
             (1, 4), (2, 5), (2, 6)]
```

Если сравнить обе структуры данных, использованные для представления нашего графа, становится ясно, что список ребер менее информативен. Это обусловлено тем, что наш граф довольно разрежен. С другой стороны, если бы наш граф был полным, нам потребовался бы 21 кортеж вместо 6. Это объясняется пространственной сложностью $O(|E|)$, где $|E|$ – это количество

ребер. Списки ребер более эффективны для хранения разреженных графов, в которых количество ребер намного меньше количества узлов.

Списки ребер чаще всего используются в задачах с большими графами. Однако проверка наличия соединения между двумя вершинами в списке ребер требует прохода по всему списку, что может занять много времени для больших графов с большим количеством ребер.

Третье и популярное представление – это **список смежности** (adjacency list). Он состоит из списка пар, в котором каждая пара представляет узел графа и смежные с ним узлы. Пары могут храниться в связном списке, словаре или других структурах данных, в зависимости от реализации. Например, список смежности для нашего графа может выглядеть так:

```
adj_list = {
    0: [1, 2],
    1: [0, 3, 4],
    2: [0, 5, 6],
    3: [1],
    4: [1],
    5: [2],
    6: [2]
}
```

Список смежности имеет несколько преимуществ перед матрицей смежности или списком ребер. Во-первых, пространственная сложность списка смежности равна $O(|V| + |E|)$, где $|V|$ – количество узлов и $|E|$ – количество ребер. Это более эффективно, чем пространственная сложность $O(|V|^2)$ матрицы смежности для разреженных графов. Во-вторых, список смежности позволяет эффективно перебирать соседние вершины узла, что полезно во многих графовых алгоритмах. Наконец, добавление узла или ребра можно выполнить за постоянное время.

Однако проверка наличия соединения между двумя вершинами при использовании списка смежности может быть медленнее, чем при использовании матрицы смежности. Это связано с тем, что для проверки требуется проход по списку, что может занять много времени для больших графов.

Каждая структура данных имеет свои преимущества и недостатки, которые зависят от конкретной области применения и требований. В следующем разделе мы поработаем с графами и познакомимся с двумя наиболее фундаментальными графовыми алгоритмами.

Изучение графовых алгоритмов

Алгоритмы на графах имеют важное значение при решении задач, связанных с графами, таких как поиск кратчайшего пути между двумя узлами или обнаружение циклов. В этом разделе мы обсудим два алгоритма обхода графа: BFS и DFS.

Поиск в ширину (breadth-first search)

BFS (Breadth-First Search) – это алгоритм обхода графа, который начинает с корневого узла и исследует все соседние узлы на определенном уровне, прежде чем перейти к следующему уровню узлов. Он работает, поддерживая очередь узлов для посещения и помечая каждый посещенный узел при добавлении его в очередь. Затем алгоритм извлекает следующий узел из очереди и исследует всех его соседей, добавляя их в очередь, если они еще не были посещены.

Работа алгоритма BFS показано на рис. 2.8.

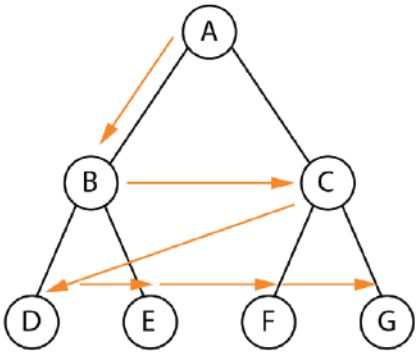


Рис. 2.8 ❖ Пример обхода графа с помощью поиска в ширину

Теперь давайте посмотрим, как мы можем реализовать поиск в ширину на Python.

1. Создаем пустой граф и добавляем ребра с помощью метода `.add_edges_from()`:

```
G = nx.Graph()
G.add_edges_from(
    [('A', 'B'), ('A', 'C'), ('B', 'D'),
     ('B', 'E'), ('C', 'F'), ('C', 'G')]
)
```

2. Мы определяем функцию под названием `bfs()`, в которой будет реализован алгоритм обхода графа BFS. Функция принимает два аргумента – объект `graph` и начальный узел для поиска:

```
def bfs(graph, node):
```

3. Инициализируем два списка (`visited` и `queue`) и добавляем начальный узел. В списке `visited` фиксируются узлы, которые были посещены во время поиска, а в списке `queue` хранятся узлы, которые необходимо посетить:

```
    visited, queue = [node], [node]
```

4. Запускаем цикл `while`, который продолжается до тех пор, пока список `queue` не станет пустым. Внутри цикла мы удаляем первый узел в списке `queue` с помощью метода `.pop(0)` и сохраняем результат в переменной `node`:

```
    while queue:
        node = queue.pop(0)
```

5. Перебираем соседей узла, используя цикл `for`. Каждого соседа, который еще не был посещен, мы добавляем в список `visited` и в конец списка `queue` с помощью метода `.append()`. По завершении возвращаем список `visited`:

```
            for neighbor in graph[node]:
                if neighbor not in visited:
                    visited.append(neighbor)
                    queue.append(neighbor)

    return visited
```

6. Вызываем функцию `bfs()`, передав в нее граф `G` и начальный узел `'A'`:

```
bfs(G, 'A')
```

7. Функция возвращает список посещенных узлов в том порядке, в котором они были посещены:

```
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

Полученный нами порядок соответствует порядку обхода графа на рис. 2.8.

BFS особенно полезен для поиска кратчайшего пути между двумя узлами невзвешенного графа. Это связано с тем, что алгоритм посещает узлы в порядке их удаленности от начального узла, поэтому при первом посещении целевого узла он должен проходить по кратчайшем пути от начального узла.

Помимо поиска кратчайшего пути, BFS еще можно использовать для проверки связности графа или для поиска всех компонент связности графа. Он также используется в таких задачах, как веб-краулинг, анализ социальных сетей и маршрутизация по кратчайшему пути в сетях.

Временная сложность BFS составляет $O(|V| + |E|)$, где $|V|$ – количество узлов и $|E|$ – количество ребер в графе. Это может быть серьезной проблемой для графов с высокой степенью связности или для разреженных графов. Для решения этой проблемы было разработано несколько вариантов BFS, например **двунаправленный BFS** (bidirectional BFS) и **A***, использующие эвристику, которая позволяет уменьшить количество узлов, необходимых для исследования.

Поиск в глубину (depth-first search)

DFS (Depth-First Search) – это рекурсивный алгоритм, который начинает работу с корневого узла и исследует, насколько это возможно, каждую ветвь, прежде чем вернуться назад.

Он выбирает узел и исследует всех его непосещенных соседей, посещая первого соседа, который не был исследован, и возвращаясь назад только тогда, когда были посещены все соседи. Таким образом, алгоритм исследует граф, проходя по как можно более глубокому пути от начального узла, прежде чем вернуться к изучению других ветвей. Это продолжается до тех пор, пока не будут исследованы все узлы.

Работа алгоритма DFS показана на рис. 2.9.

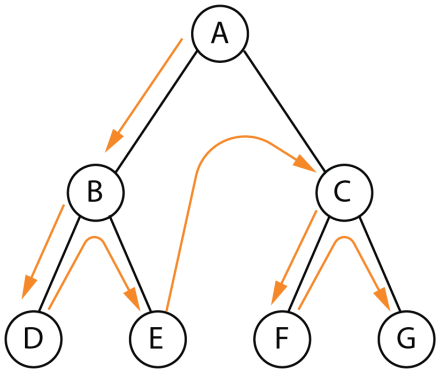


Рис. 2.9 ❖ Пример обхода графа с помощью поиска в глубину

Давайте реализуем поиск в глубину на Python.

1. Сначала мы инициализируем пустой список `visited`:

```
visited = []
```

2. Определяем функцию под названием `dfs()`, в которой будет реализован алгоритм обхода графа DFS. Функция принимает три аргумента – пустой список `visited`, объект `graph` и начальный узел для поиска `node`:

```
def dfs(visited, graph, node):
```

3. Если текущего узла нет в списке `visited`, добавляем его в список:

```
    if node not in visited:
        visited.append(node)
```

4. Затем перебираем каждого соседа текущего узла `node`. Для каждого соседа мы рекурсивно вызываем функцию `dfs()`, передавая `visited`, `graph` и `neighbor` в качестве аргументов:

```
            for neighbor in graph[node]:
                visited = dfs(visited, graph, neighbor)
```

5. Функция `dfs()` продолжает исследовать граф в глубину, посещая всех соседей каждого узла до тех пор, пока не останется непосещенных соседей. Наконец, она возвращает список `visited`:

```
    return visited
```

6. Мы вызываем функцию `dfs()`, передав ей пустой список `visited`, граф `G` и начальный узел `'A'`:

```
dfs(visited, G, 'A')
```

7. Функция возвращает список посещенных узлов в том порядке, в котором они были посещены:

```
['A', 'B', 'D', 'E', 'C', 'F', 'G']
```

Полученный нами порядок соответствует порядку обхода графа на рис. 2.9.

DFS полезен при решении различных задач, таких как поиск компонент связности, топологическая сортировка и решения задач о лабиринтах. Он особенно полезен для поиска циклов в графе, поскольку обходит граф в глубину, а цикл существует тогда и только тогда, когда узел посещается дважды во время обхода.

Как и BFS, его временная сложность равна $O(|V| + |E|)$, где $|V|$ – количество узлов и $|E|$ – количество ребер в графе. Он требует меньше памяти, но не гарантирует решение задачи о кратчайшем пути. Наконец, в отличие от BFS, при использовании DFS можно застрять в бесконечных циклах.

Кроме того, многие другие алгоритмы в теории графов основаны на BFS и DFS, такие как алгоритм кратчайшего пути Дейкстры, алгоритм минимального остовного дерева Крускала и алгоритм поиска компонент сильной связности Тарьяна. Таким образом, глубокое понимание BFS и DFS необходимо для всех, кто хочет работать с графами и разрабатывать более сложные графовые алгоритмы.

Выводы

В этой главе мы рассмотрели основы теории графов – раздела математики, изучающего графы и сети. Мы начали с определения графа и рассказали о различных типах графов (ориентированные, взвешенные и связные графы). Затем представили фундаментальные объекты графа (включая соседей) и меры (такие как центральность и плотность), которые используются для объяснения и анализа структур графа.

Кроме того, мы обсудили различные способы представления графа. Наконец, мы исследовали два фундаментальных алгоритма обхода графа – BFS

и DFS, которые составляют основу для разработки более сложных графовых алгоритмов.

В главе 3 «Создание описаний узлов с помощью DeepWalk» мы рассмотрим архитектуру DeepWalk и два ее компонента: Word2Vec и случайные блуждания. Начнем с объяснения архитектуры Word2Vec, а затем реализуем ее с помощью специализированной библиотеки. Затем мы углубимся в алгоритм DeepWalk и реализуем случайные блуждания по графу.

ЧАСТЬ II

ОСНОВЫ

Во второй части книги мы глубже рассмотрим процесс построения представлений узлов с использованием обучения на графах. Начнем с изучения традиционных методов обучения на графах, опираясь на достижения в области обработки естественного языка. Наша цель – понять, как эти методы можно применить к графам и как их можно использовать для построения представлений узлов.

Затем мы перейдем к включению информации, описывающей узлы, в наши модели и посмотрим, как их можно использовать для создания еще более точных представлений. Наконец, познакомимся с двумя наиболее фундаментальными архитектурами GNN: **графовой сверточной нейронной сетью** (Graph Convolutional Network – GCN) и **графовой нейронной сетью с механизмом внимания** (Graph Attention Network – GAT). Эти две архитектуры являются строительными блоками многих современных методов обучения на графах и обеспечат прочную основу для понимания следующей части книги.

К концу этой части вы получите более глубокое понимание того, как традиционные методы обучения на графах, такие как случайные блуждания, могут использоваться для создания представлений узлов и решения задач. Кроме того, вы узнаете, как создавать еще более мощные представления с помощью GNN. Вы познакомитесь с двумя ключевыми архитектурами GNN и узнаете, как их можно использовать для решения различных задач на основе графов.

Эта часть состоит из следующих глав:

- глава 3 «Создание представлений узлов с помощью DeepWalk»;
- глава 4 «Улучшение эмбеддингов с помощью смещенных случайных блужданий в Node2Vec»;
- глава 5 «Включение информации о характеристиках узлов с помощью простых нейронных сетей»;
- глава 6 «Знакомство со графовыми сверточными сетями»;
- глава 7 «Графовые нейронные сети с механизмом внимания».

Глава 3

Создание представлений узлов с помощью DeepWalk

DeepWalk – один из первых крупных успешных примеров применения методов машинного обучения (ML) для графовых данных. Он знакомит нас с такими важными понятиями, как эмбеддинги, которые лежат в основе GNN. В отличие от традиционных нейронных сетей цель этой архитектуры – создавать **описания** или **представления** (representations), которые затем передаются в другие модели, выполняющие последующие задачи (например, классификацию узлов).

В этой главе мы изучим архитектуру DeepWalk и два ее основных компонента: Word2Vec и **случайные блуждания** (random walks). Мы объясним, как работает архитектура Word2Vec, уделив особое внимание модели Skip-gram. Мы реализуем эту модель с помощью популярной библиотеки gensim на примере **обработки естественного языка** (natural language processing – NLP), чтобы понять, как ее следует использовать.

Затем мы сосредоточимся на алгоритме DeepWalk и посмотрим, как можно повысить качество с помощью иерархической функции softmax (hierarchical softmax – H-Softmax). Эту мощную оптимизацию функции softmax можно найти во многих областях: она невероятно полезна, когда в вашей задаче классификации большое количество возможных классов. Мы также реализуем случайные блуждания по графу, а затем завершим главу классификацией с учителем, примененной к набору данных Клуба карате Закари.

К концу этой главы вы освоите Word2Vec в контексте NLP и за его пределами. Вы сможете создавать эмбеддинги узлов, используя топологическую информацию графов, и решать задачи классификации на основе графовых данных.

В этой главе будут рассмотрены следующие основные темы:

- «Знакомство с Word2Vec»,
- «DeepWalk и случайные блуждания»,
- «Реализация DeepWalk».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter03.

Знакомство с Word2Vec

Первым шагом к освоению алгоритма DeepWalk является понимание его основного компонента – Word2Vec.

Word2Vec стал одним из самых важных методов глубокого обучения в NLP. Метод был опубликован в 2013 году Томасом Миколовым совместно с коллегами (Google) в двух разных статьях, в нем была предложена новая техника перевода слов в векторы (также известная как **эмбеддинги** (embeddings)) с использованием больших наборов текстовых данных. Затем эти представления можно использовать в таких задачах, как классификация тональности текста. Кроме того, Word2Vec представляет собой один из редких примеров запатентованной и популярной архитектуры машинного обучения.

Вот несколько примеров того, как Word2Vec может преобразовывать слова в векторы:

$$vec(king) = [-2.1, 4.1, 0.6]$$
$$vec(queen) = [-1.9, 2.6, 1.5]$$
$$vec(man) = [3.0, -1.1, -2]$$
$$vec(woman) = [2.8, -2.6, -1.1].$$

В этом примере мы видим, что с точки зрения евклидова расстояния векторы слов для *king* и *queen* ближе друг к другу, чем векторы слов для *king* и *woman* (1.76 против 8.47).

```
# импортируем необходимые библиотеки
import numpy as np
import math

# создаем векторы слов king, woman и queen
vec_king = np.array([-2.1, 4.1, 0.6])
vec_woman = np.array([2.8, -2.6, -1.1])
vec_queen = np.array([-1.9, 2.6, 1.5])

# пишем функцию, вычисляющую евклидово расстояние
# между двумя векторами слов
def euclidean_distance(x1, x2):
    """
    Вычисляет евклидово расстояние
    между векторами слова.
    """
    distance = 0
```

```
    for i in range(len(x1)):
        distance += pow((x1[i] - x2[i]), 2)
    return math.sqrt(distance)

# вычисляем евклидово расстояние
# между векторами слов king и queen
print(euclidean_distance(vec_king, vec_queen))
# вычисляем евклидово расстояние
# между векторами слов king и woman
print(euclidean_distance(vec_king, vec_woman))
```

```
1.7606816861659005
8.472897969408105
```

Обычно для измерения сходства этих слов используются другие метрики, например популярное **косинусное сходство** (cosine similarity). Косинусное сходство фокусируется на угле между векторами и не учитывает их величину (длину), что более полезно при их сравнении. Косинусное сходство вычисляется по формуле:

$$cosine\ similarity(\vec{A}, \vec{B}) = \cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \cdot \|\vec{B}\|}.$$

Одним из самых удивительных результатов Word2Vec является его способность решать аналогии. Популярный пример – как Word2Vec может ответить на вопрос «*man is to woman, what king is to __?*» («мужчина для женщины то же, что король для __?»). Ответ можно вычислить следующим образом:

$$vec(king) - vec(man) + vec(woman) \approx vec(queen).$$

Это утверждение не применимо ко всем аналогиям, но такая особенность может привести к интересным примерам применения для выполнения арифметических операций с эмбеддингами.

CBOW против skip-gram

Модель должна быть обучена на предтекстовой задаче с целью создания этих векторов. Сама задача не обязательно должна быть содержательной: ее единственная цель – создание высококачественных эмбеддингов. На практике эта задача всегда связана с предсказанием слов в определенном контексте.

Авторы предложили две архитектуры со схожими задачами.

- **Непрерывный мешок слов** (continuous bag-of-words (CBOW) model): она обучается, чтобы прогнозировать слово, используя его окружающий контекст (слова, идущие до и после целевого слова). Порядок контекстных слов не имеет значения, поскольку их эмбеддинги суммируются в модели. Авторы утверждают, что получают лучшие результаты, используя четыре слова до и после предсказанного.

- **Непрерывный skip-gram** (continuous skip-gram model). Здесь мы вводим в модель отдельное слово и пытаемся предсказать слова вокруг него. Увеличение диапазона контекстных слов приводит к получению лучших эмбеддингов, но также увеличивает время обучения.

Итак, в модели CBOW мы получаем эмбеддинги, обучая модель предсказывать слова по контексту, а в модели skip-gram мы получаем эмбеддинги, обучая модель предсказывать контекст по слову.

Подведем итог, ниже на рис. 3.1 представлены входной и выходной слои обеих моделей.

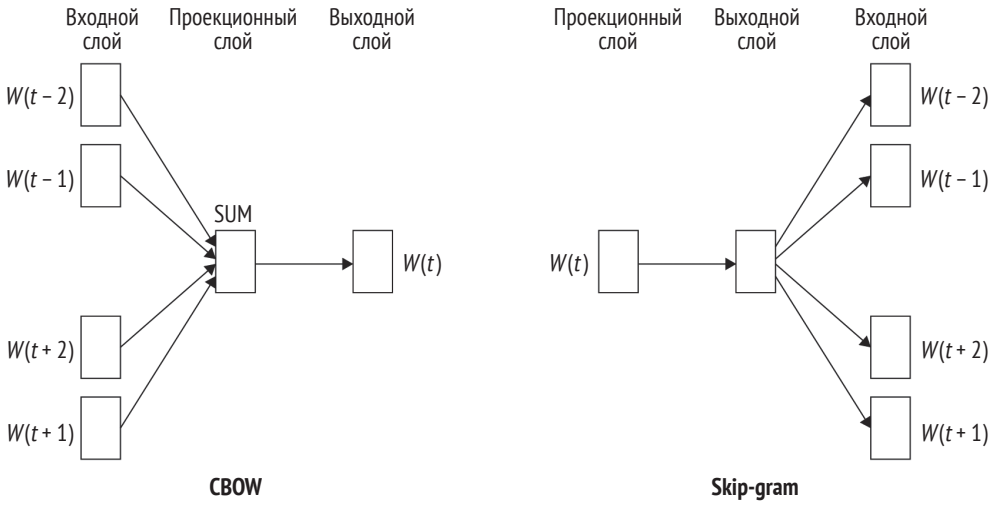


Рис. 3.1 ❖ Архитектуры CBOW и skip-gram

В целом модель CBOW считается более быстрой в плане обучения, но модель skip-gram более точна благодаря способности запоминать редкие слова. Эта тема до сих пор обсуждается в NLP-сообществе: различные реализации могут решить проблемы, связанные с применением CBOW в тех или иных аспектах.

Создание скип-грамм

На данный момент мы сосредоточимся на модели skip-gram, поскольку именно эту архитектуру использует DeepWalk. Скип-граммы реализованы как пары слов со следующей структурой: (*target word*, *context word*), где *target word* (*целевое слово*) – входное слово, а *context word* (*контекстное слово*) – слово, которое нужно спрогнозировать. Количество скип-грамм для одного и того же целевого слова зависит от параметра, называемого **размером контекста** (context size), как показано на рис. 3.2.

Размер контекста	Текст	Скип-граммы
1	**the *train*** was late.	('the', 'train')
	the* train *was late	('train', 'the') ('train', 'was')
	the ***train* was *late***	('was', 'train') ('was', 'late')
	the train ***was* late**	('late', 'was')
2	**the *train was*** late	('the', 'train') ('the', 'was')
	the* train *was late	('train', 'the') ('train', 'was') ('train', 'late')
	the train* was *late	('was', 'the') ('was', 'train') ('was', 'late')
	the ***train was* late**	('late', 'train') ('late', 'was')

Рис. 3.2 ❖ Текст для скип-грамм

Ту же самую идею можно применить к корпусу текста, а не к одному предложению.

На практике мы сохраняем все контекстные слова для одного и того же целевого слова в списке для экономии памяти. Давайте посмотрим, как это делается, на примере целого абзаца.

В следующем примере мы создаем скип-граммы для всего абзаца, хранящегося в текстовой переменной. Мы устанавливаем для переменной `CONTEXT_SIZE` значение 2, это означает, что мы будем смотреть на два слова до и после целевого слова.

1. Сначала задаем стартовое значение генератора псевдослучайных чисел:

```
np.random.seed(0)
```

2. Затем нам нужно установить для переменной `CONTEXT_SIZE` значение 2 и ввести текст, который мы хотим проанализировать:

```
CONTEXT_SIZE = 2

text = """Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc eu sem
scelerisque, dictum eros aliquam, accumsan quam. Pellentesque tempus, lorem ut
semper fermentum, ante turpis accumsan ex, sit amet ultricies tortor erat quis
nulla. Nunc consectetur ligula sit amet purus porttitor, vel tempus tortor
scelerisque. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices
posuere cubilia curae; Quisque suscipit ligula nec faucibus accumsan. Duis
vulputate massa sit amet viverra hendrerit. Integer maximus quis sapien id
convallis. Donec elementum placerat ex laoreet gravida. Praesent quis enim
```

```
facilisis, bibendum est nec, pharetra ex. Etiam pharetra congue justo, eget
imperdiet diam varius non. Mauris dolor lectus, interdum in laoreet quis,
faucibus vitae velit. Donec lacinia dui eget maximus cursus. Class aptent taciti
sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Vivamus
tincidunt velit eget nisi ornare convallis. Pellentesque habitant morbi
tristique senectus et netus et malesuada fames ac turpis egestas. Donec
tristique ultrices tortor at accumsan.
""".split()
```

3. Далее создаем скип-граммы с помощью простого цикла `for`, позволяющего учитывать каждое слово в тексте. С помощью генератора списка создаем контекстные слова, хранящиеся в списке `skipgrams`:

```
# создаем скип-граммы
skipgrams = []
for i in range(CONTEXT_SIZE, len(text) - CONTEXT_SIZE):
    array = [text[j] for j in np.arange(i - CONTEXT_SIZE,
                                        i + CONTEXT_SIZE + 1) if j != i]
    skipgrams.append((text[i], array))
```

4. Наконец, воспользуемся функцией `print()`, чтобы посмотреть сгенерированные скип-граммы:

```
print(skipgrams[0:2])
```

5. Она печатает следующий вывод:

```
[('dolor', ['Lorem', 'ipsum', 'sit', 'amet,']), ('sit', ['ipsum', 'dolor', 'amet,',
'consectetur'])]
```

Эти два целевых слова с соответствующим контекстом показывают, как выглядят входные данные для Word2Vec.

Модель skip-gram

Цель Word2Vec – создать высококачественные эмбеддинги слов. Чтобы получить эти эмбеддинги, задача обучения модели skip-gram состоит в предсказании правильных контекстных слов для данного целевого слова.

Представьте, что у нас есть последовательность из N слов w_1, w_2, ..., w_N. Вероятность встретить слово w_2 при наличии слова w_1 можно записать как $p(w_2|w_1)$. Наша цель – максимизировать сумму каждой вероятности встретить контекстное слово для данного целевого слова во всем тексте:

$$\frac{1}{N}\sum_{n=1}^{N}\sum_{-c\le j\le c,\ j\neq 0}\log p(w_{n+j}|w_n),$$

где C – размер вектора контекста.

Примечание

Почему мы используем логарифмическую вероятность в предыдущем уравнении? Преобразование вероятностей в логарифмические вероятности является распространенным методом в ML (и в информатике в целом) по двум основным причинам.

Операции умножения становятся операциями сложениями (а операции деления – операциями вычитания). Умножение требует бóльших вычислительных затрат, чем сложение, поэтому логарифмическую вероятность вычислить быстрее:

$$\log(A \times B) = \log(A) + \log(B).$$

Способ, с помощью которого компьютеры хранят очень маленькие числа (например, 3.14e-128), не является абсолютно точным в отличие от логарифма тех же чисел (в данном случае –127.5). Эти небольшие ошибки могут суммироваться и искажать конечные результаты, когда события крайне маловероятны.

В целом это простое преобразование позволяет нам увеличить скорость и точность, не меняя нашей первоначальной цели.

Базовая модель skip-gram использует функцию softmax для вычисления вероятности встретить эмбеддинг контекстного слова при заданном эмбеддинге целевого слова:

$$p(w_c|w_t) = \frac{\exp(h_c h_t^T)}{\sum_{i=1}^{|V|} \exp(h_i h_t^T)},$$

где V – словарь размера $|V|$. Этот словарь соответствует списку уникальных слов, которые модель пытается предсказать. Мы можем получить этот список, используя множество для удаления повторяющихся слов:

```
vocab = set(text)
VOCAB_SIZE = len(vocab)
print(f"Длина словаря = {VOCAB_SIZE}")
```

Это дает нам следующий вывод:

```
Длина словаря = 121
```

Теперь, когда у нас есть размер словаря, нужно определить еще один параметр: N, размерность векторов слов. Обычно это значение устанавливается в диапазоне от 100 до 1000. В этом примере мы установим значение 10 из-за ограниченного размера нашего набора данных.

Модель skip-gram состоит всего из двух слоев.

- **Проекционный слой** (projection layer) содержит матрицу весов W_{embed}, принимает в качестве входа one-hot-закодированный вектор слова и возвращает соответствующий N-мерный эмбеддинг слова. Он работает как простая таблица поиска, в которой хранятся эмбеддинги заранее заданной размерности.

- **Полносвязный слой** (fully connected layer) содержит весовую матрицу W_{output}, принимает в качестве входа эмбеддинг слова и возвращает $|V|$-мерные логиты. К этим прогнозам применяется функция softmax для преобразования логитов в вероятности.

> **Примечание**
>
> Здесь не используется функция активации. Word2Vec – это линейный классификатор, моделирующий линейные отношения между словами.

Пусть x – one-hot-закодированный вектор слова, играющий роль *входа* (*input*). Соответствующий эмбеддинг слова можно вычислить как простую проекцию:

$$h = W_{embed}^{T} \cdot x.$$

Используя модель skip-gram, мы можем переписать ранее сформулированную вероятность следующим образом:

$$p(w_c|w_t) = \frac{\exp(w_{output} \cdot h)}{\sum_{i=1}^{|V|} \exp(w_{output_{(i)}} \cdot h)}.$$

Модель skip-gram выдает $|V|$-мерный вектор, который является условной вероятностью каждого слова в словаре:

$$word2vec(w_t) = \begin{bmatrix} p(w_1|w_t) \\ p(w_2|w_t) \\ \vdots \\ p(w_{|V|}|w_t) \end{bmatrix}.$$

Во время обучения эти вероятности сравниваются с правильными one-hot-закодированными векторами целевых слов. Разница между этими значениями (рассчитанная с помощью такой функции потерь, как кросс-энтропия) передается обратно по сети для обновления весов и получения более точных прогнозов.

Вся архитектура Word2Vec представлена на рис. 3.3 с обеими матрицами и финальным слоем с функцией softmax.

Мы можем реализовать эту модель с помощью библиотеки gensim, которая также используется в официальной реализации DeepWalk. Затем можем создать словарь и обучить нашу модель на основе предыдущего текста.

1. Начнем с установки библиотеки gensim и импорта класса `Word2Vec`:

```
!pip install -qU gensim
from gensim.models.word2vec import Word2Vec
```

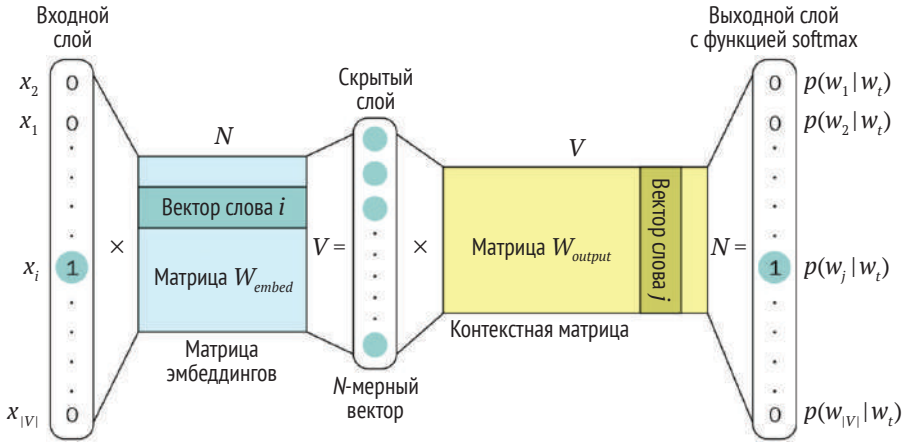


Рис. 3.3 ❖ Архитектура Word2Vec

2. Мы инициализируем модель skip-gram с помощью объекта `Word2Vec` и параметра `sg=1` (skip-gram = 1):

```
# создаем Word2Vec
model = Word2Vec([text],
                 # 0 - CBOW, 1 - skip-gram
                 sg=1,
                 vector_size=10,
                 min_count=0,
                 window=2,
                 workers=1,
                 seed=0)
```

3. Теперь проверим форму нашей первой матрицы весов. Она должна соответствовать размеру словаря и размерности эмбеддингов слов:

```
print(f'Форма матрицы W_embed: {model.wv.vectors.shape}')
```

4. Получаем следующий вывод:

```
Форма матрицы W_embed: (121, 10)
```

5. Затем обучаем модель в течение 10 эпох:

```
# обучаем модель
model.train([text],
            total_examples=model.corpus_count,
            epochs=10)
```

6. Наконец, печатаем эмбеддинг слова, чтобы увидеть, как выглядит результат нашего обучения:

```
# печатаем эмбеддинг слова
print('\nЭмбеддинг слова =')
print(model.wv[0])
```

7. Получаем следующий вывод:

```
Эмбеддинг слова =
[ 0.07156403  0.03257632  0.00209916 -0.04374931 -0.03398107 -0.08656936
 -0.09047253 -0.0955243  -0.06482638  0.0660186 ]
```

Хотя этот подход хорошо работает с небольшими словарями, вычислительные затраты, связанные с применением полной функции softmax к миллионам слов (размер словаря), в большинстве случаев будут слишком высокими. Данное обстоятельство долгое время было ограничивающим фактором в разработке точных языковых моделей. К счастью для нас, для решения этой проблемы были разработаны другие подходы.

В Word2Vec (и DeepWalk) реализован один из этих подходов под названием **иерархический softmax** (hierarhical softmax – H-softmax). Вместо плоского softmax (flat softmax), который напрямую вычисляет вероятность каждого слова, этот метод использует структуру бинарного дерева, в котором листьями являются слова. Еще более интересно то, что можно использовать дерево Хаффмана, в котором редкие слова хранятся на более глубоких уровнях, чем обычные слова. В большинстве случаев это значительно ускоряет предсказание слов как минимум в 50 раз.

В библиотеке gensim H-Softmax можно активировать, выбрав `hs=1`.

Это была самая сложная часть архитектуры DeepWalk. Однако перед тем, как реализовать ее, нам нужно выяснить, как создавать обучающие данные для нее.

DeepWalk и случайные блуждания

Предложенный в 2014 году Perozzi et al., DeepWalk быстро стал чрезвычайно популярным алгоритмом среди исследователей графов. Вдохновленный недавними достижениями в области NLP, он неизменно превосходил другие алгоритмы на различных наборах данных. Хотя с тех пор были предложены более производительные архитектуры, DeepWalk представляет собой простую и надежную модель, которую можно быстро внедрить для решения множества проблем.

Цель DeepWalk – создавать высококачественные векторные (признаковые) представления узлов в режиме обучения без учителя. Эта архитектура во многом была вдохновлена Word2Vec в NLP. Однако вместо слов наш набор данных состоит из узлов. Вот почему мы используем случайные блуждания для создания содержательных последовательностей узлов, которые выполняют роль предложений. Диаграмма на рис. 3.4 иллюстрирует связь между предложениями и графами.

Случайные блуждания – это последовательности узлов, создаваемые случайным выбором соседнего узла на каждом шаге. С помощью них мы получим признаковые представления узлов нашего графа. По сути, это будут

числовые векторы, созданные таким образом, чтобы сохранить структурные и свойственные узлам графа характеристики.

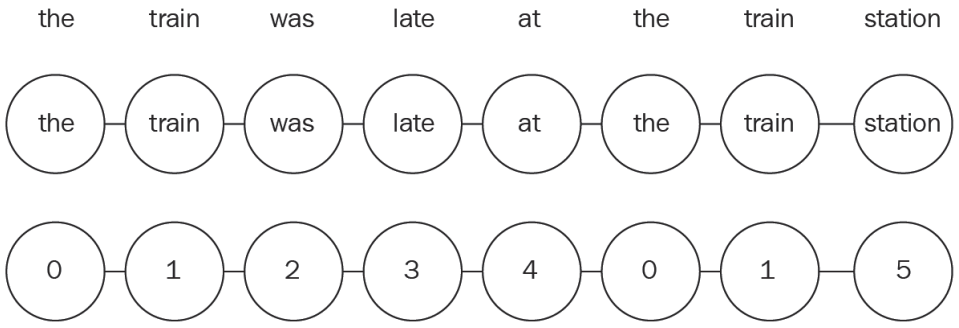


Рис. 3.4 ❖ Предложения можно представить в виде графов

Например, узлы могут появляться несколько раз в одной и той же последовательности. Почему это важно? Даже если узлы выбраны случайным образом, тот факт, что они часто появляются вместе в последовательности, означает, что они расположены близко друг к другу. Согласно гипотезе о гомофилии сети, узлы, расположенные близко друг к другу, схожи друг с другом. Это особенно актуально в социальных сетях, где люди связаны с друзьями и семьей.

Эта идея лежит в основе алгоритма DeepWalk: когда узлы расположены близко друг к другу, мы хотим получить высокие оценки сходства. Напротив, когда узлы находятся далеко друг от друга, нам нужны низкие оценки сходства.

Давайте реализуем функцию случайного блуждания, используя граф networkx.

1. Импортируем необходимые библиотеки:

```
import networkx as nx
import matplotlib.pyplot as plt
```

2. Мы генерируем случайный граф благодаря функции `erdos_renyi_graph()` с фиксированным количеством узлов (10) и заранее определенной вероятностью появления ребра между двумя узлами (0, 3):

```
# создаем граф
G = nx.erdos_renyi_graph(10, 0.3, seed=1, directed=False)
```

3. Строим этот случайный граф и смотрим его:

```
# визуализируем граф
plt.figure()
plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
```

```
node_size=600,
cmap='coolwarm',
font_size=14,
font_color='white'
)
```

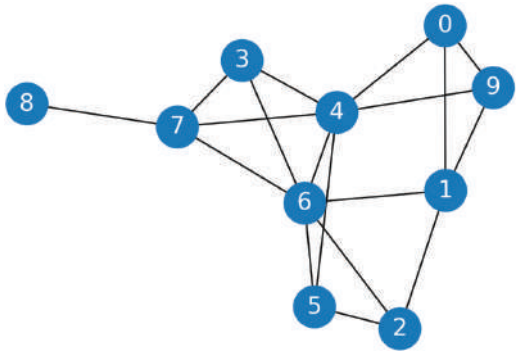


Рис. 3.5 ❖ Случайный граф

4. Теперь задаем стартовое значение генератора псевдослучайных чисел и реализуем случайные блуждания с помощью простой функции `random_walk()`. Эта функция принимает два параметра: начальный узел (`start`) и длину блуждания (`length`). На каждом шаге мы случайным образом выбираем соседний узел (используя `np.random.choice`) до завершения блуждания:

```
np.random.seed(0)

def random_walk(start, length):
    walk = [str(start)]  # стартовый узел

    for i in range(length):
        neighbors = [node for node in G.neighbors(start)]
        next_node = np.random.choice(neighbors, 1)[0]
        walk.append(str(next_node))
        start = next_node

    return walk
```

5. Далее выводим результат этой функции – список с начальным узлом `0` и длиной `10`:

```
# получаем результат случайных блужданий
print(random_walk(0, 10))
```

```
['0', '17', '1', '17', '0', '4', '6', '16', '6', '5', '0']
```

В таких последовательностях мы можем увидеть, что определенные узлы часто встречаются вместе. Данный факт означает, что эти узлы схожи, учи-

тывая, что мы работаем с гомофильным графом. Именно такие отношения мы пытаемся уловить с помощью DeepWalk.

Теперь, когда мы реализовали Word2Vec и случайное блуждание по отдельности, давайте объединим их, чтобы создать DeepWalk.

Реализация DeepWalk

Теперь, когда мы хорошо понимаем каждый компонент этой архитектуры, давайте воспользуемся им для решения задачи машинного обучения.

Набор данных, который мы будем использовать, – это Клуб карате Закари. Он просто представляет отношения между участниками Клуба карате, который изучал Уэйн В. Закари в 1970-х годах. Это своего рода социальная сеть, в которой каждый узел является участником, а участники, взаимодействующие друг с другом за пределами клуба, связаны между собой.

В этом примере данные об участниках разделены на два класса: мы хотели бы присвоить правильный класс каждому участнику (классификация узлов), просто посмотрев его связи с другими участниками.

1. Давайте загрузим набор данных с помощью функции `nx.karate_club_graph()`:

```
# загрузим набор данных
G = nx.karate_club_graph()
```

2. Далее нам нужно преобразовать строковые метки классов в целочисленные значения 0 и 1 (`Mr. Hi = 0`, `Officer = 1`):

```
# получаем метки (Mr. Hi = 0, Officer = 1)
labels = []
for node in G.nodes:
    label = G.nodes[node]['club']
    labels.append(1 if label == 'Officer' else 0)
```

3. Давайте визуализируем этот граф, используя наши новые метки:

```
# визуализируем граф
plt.figure(figsize=(12,12))
plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 node_color=labels,
                 node_size=800,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```

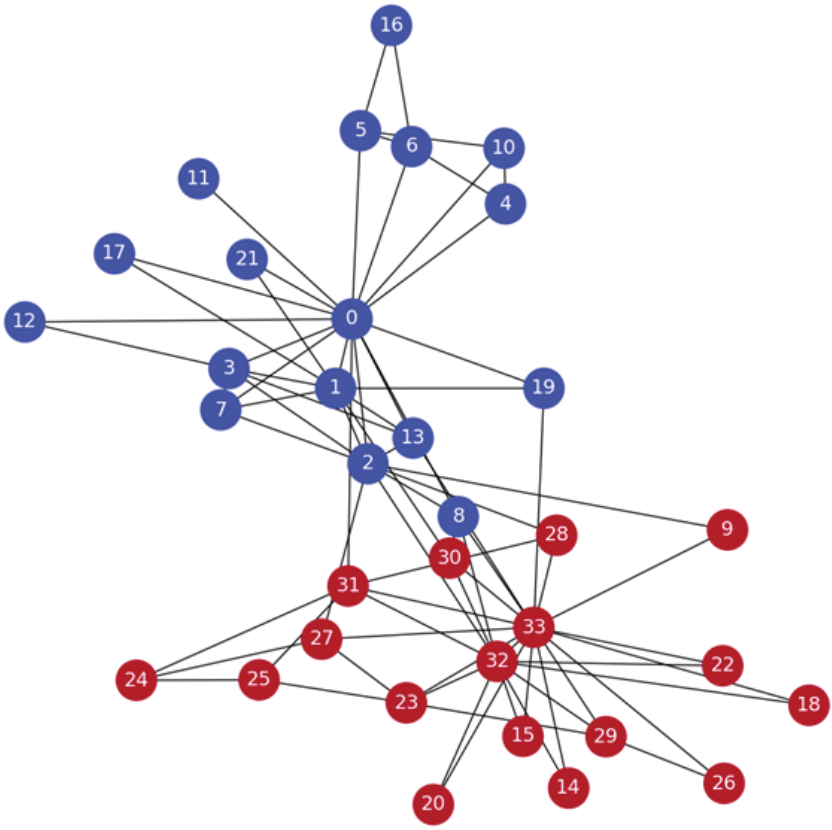


Рис. 3.6 ❖ Клуб карате Закари

4. Следующим шагом будет создание набора случайных блужданий. Мы стремимся к максимальной полноте описания свойств графа, поэтому создадим 80 случайных блужданий длиной 10 для каждого узла графа:

```
# создаем список, в котором будут записаны
# результаты случайных блужданий
walks = []
for node in G.nodes:
    for _ in range(80):
        walks.append(random_walk(node, 10))
```

5. Напечатаем первое сгенерированное случайное блуждание:

```
# напечатаем первое случайное блуждание
print(walks[0])
```

```
['0', '10', '0', '17', '0', '2', '13', '0', '2', '9', '33']
```

6. Последний шаг заключается в реализации Word2Vec. Здесь мы инициализируем модель skip-gram с иерархической функцией softmax. Вы

можете поэкспериментировать с другими параметрами, чтобы улучшить качество эмбеддингов:

```
# создаем Word2Vec
model = Word2Vec(walks,
                 hs=1, # иерархический softmax
                 sg=1, # модель skip-gram
                 vector_size=100,
                 window=10,
                 workers=1,
                 seed=1)

print(f'Форма матрицы эмбеддингов: {model.wv.vectors.shape}')
```

```
Форма матрицы эмбеддингов: (34, 100)
```

7. Затем мы просто обучаем модель на сгенерированных нами случайных блужданиях.

```
# создаем словарь
model.build_vocab(walks)

# обучаем модель
model.train(walks,
            total_examples=model.corpus_count,
            epochs=30,
            report_delay=1);
```

8. Теперь, когда наша модель обучена, давайте посмотрим на различные примеры ее применения. Первый из них – поиск узлов, наиболее близких к заданному узлу (с точки зрения косинусного сходства):

```
# самые схожие узлы
print('Узлы, которые больше всего похожи на узел 0:')
for similarity in model.wv.most_similar(positive=['0']):
    print(f'{similarity}')
```

```
Узлы, которые больше всего похожи на узел 0:
('7', 0.6418899893760681)
('11', 0.6362671256065369)
('10', 0.6353005766868591)
('4', 0.6283879280090332)
('1', 0.6240326762199402)
('17', 0.6081677675247192)
('6', 0.5763440132141113)
('5', 0.5598748326301575)
('21', 0.5572237372398376)
('16', 0.550387442111969)
```

Еще одно важное применение – вычисление оценки сходства между двумя узлами. Эту оценку можно вычислить следующим образом:

```
# сходство между двумя узлами
print(f"\nСходство между узлами 0 и 4: {model.wv.similarity('0', '4')}")
```

```
Сходство между узлами 0 и 4: 0.6283879280090332
```

Здесь мы получаем косинусное сходство между двумя узлами.

Мы можем визуализировать полученные эмбеддинги. Давайте воспользуемся **стохастическим вложением соседей с *t*-распределением** (t-distributed stochastic neighbor embedding – t-SNE), чтобы визуализировать полученные высокоразмерные векторы в 2D.

1. Импортируем класс `TSNE` библиотеки sklearn:

```
from sklearn.manifold import TSNE
```

2. Создаем два массива: один – для хранения эмбеддингов слов, а другой – для хранения меток:

```
# подготовим векторы слов и метки
nodes_wv = np.array([model.wv.get_vector(str(i))
                     for i in range(len(model.wv))])
labels = np.array(labels)
```

3. Затем мы обучаем на эмбеддингах модель t-SNE с размерностью 2 (`n_components=2`):

```
# обучаем TSNE
tsne = TSNE(n_components=2,
            learning_rate='auto',
            init='pca',
            random_state=0).fit_transform(nodes_wv)
```

4. Наконец, давайте построим двумерные векторы, созданные обученной моделью t-SNE, с соответствующими метками:

```
# визуализируем TSNE
plt.figure(figsize=(6, 6))
plt.scatter(tsne[:, 0], tsne[:, 1], s=100, c=labels, cmap="coolwarm")
plt.show()
```

Этот график весьма обнадеживает, поскольку мы видим четкую линию, разделяющую два класса. Простой алгоритм машинного обучения может классифицировать эти узлы с помощью достаточного количества наблюдений (обучающих данных). Давайте реализуем классификатор и обучим его на наших эмбеддингах узлов.

1. Мы импортируем модель случайного леса из sklearn, которая является популярным выбором, когда дело доходит до классификации. Правильность (accuracy) – это метрика, которую мы будем использовать для оценки этой модели:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

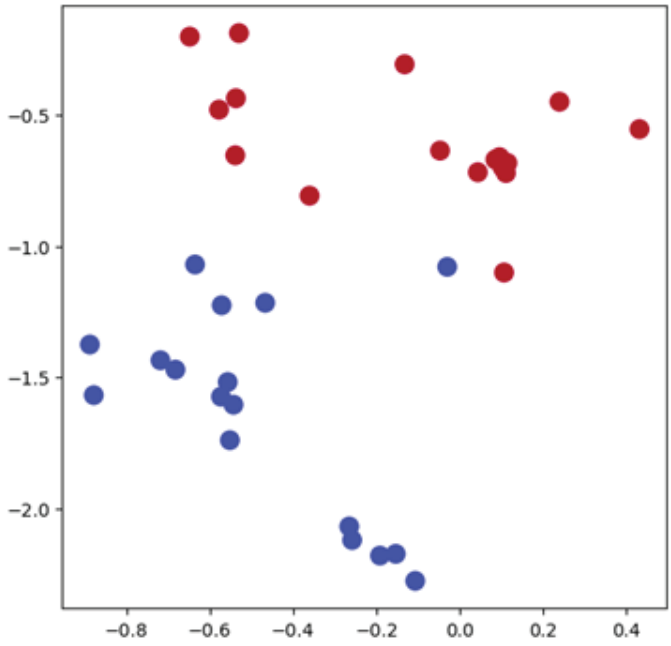


Рис. 3.7 ❖ Представление узлов с помощью t-SNE

2. Нам нужно разделить эмбеддинги на две группы: обучающие и тестовые данные. Простой способ сделать это – воспользоваться предварительно созданными масками:

```
# создаем маски для обучения и теста
train_mask = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
test_mask = [0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21,
             23, 25, 26, 27, 28, 29, 30, 31, 32, 33]
```

3. Затем мы обучаем классификатор – модель случайного леса на обучающих данных с соответствующими метками:

```
# обучаем классификатор
clf = RandomForestClassifier(random_state=0)
clf.fit(nodes_wv[train_mask], labels[train_mask])
```

4. Наконец, оцениваем обученную модель на тестовых данных с точки зрения правильности:

```
# оцениваем качество
y_pred = clf.predict(nodes_wv[test_mask])
acc = accuracy_score(y_pred, labels[test_mask])
print(f'Правильность = {acc*100:.2f}%')
```

5. Вот итоговое качество нашего классификатора:

```
Правильность = 90.91%
```

Наша модель получила оценку правильности 95.45 %, что довольно неплохо, учитывая разбиение на обучающую и тестовую выборки, которое мы ей дали. Есть еще возможности для улучшения, но в этом примере мы просто показали два полезных примера применения DeepWalk:

- *обнаружение сходства между узлами* с использованием эмбеддингов и косинусного сходства (обучение без учителя);
- *использование этих эмбеддингов в качестве набора данных* для таких задач обучения с учителем, как классификация узлов.

Как мы увидим в следующих главах, возможность изучения представлений узлов дает большую гибкость при проектировании более глубоких и сложных архитектур.

Выводы

В этой главе мы узнали об архитектуре DeepWalk и ее основных компонентах. Затем преобразовали данные графа в последовательности, используя случайные блуждания, чтобы применить мощный алгоритм Word2Vec. Полученные эмбеддинги можно использовать для поиска сходства между узлами или в качестве входных данных для других алгоритмов. В частности, мы решили задачу классификации узлов, используя обучение с учителем.

В главе 4 «Улучшение эмбеддингов с помощью смещенных случайных блужданий в Node2Vec» мы представим второй алгоритм, основанный на Word2Vec. Отличие от DeepWalk заключается в том, что при генерации случайных блужданий некоторые узлы могут быть «исследованы» в большей или меньшей степени, что напрямую влияет на создаваемые эмбеддинги. Мы реализуем этот алгоритм на новом примере и сравним его представления с представлениями, полученными с помощью DeepWalk.

Дополнительное чтение

[1] F. Xia et al., *Graph Learning*: A Survey, IEEE Transactions on Artificial Intelligence, vol. 2, no. 2, pp. 109–127, Apr. 2021, DOI:10.1109/tai.2021.3076021. Доступ по ссылке https://arxiv.org/abs/2105.00696.

[2] A. Trafton, *Artificial intelligence yields new antibiotic*, MIT News, 20-Feb-2020. [Online]. Доступ по ссылке https://news.mit.edu/2020/artificial-intelligence-identifies-new-antibiotic-0220.

Глава 4

Улучшение эмбеддингов с помощью смещенных случайных блужданий в Node2Vec

Node2Vec – это архитектура, во многом основанная на DeepWalk. В предыдущей главе мы рассмотрели два основных компонента этой архитектуры – случайные блуждания и Word2Vec. Как мы можем улучшить качество наших эмбеддингов? Интересно, что это можно выполнить, не прибегая к машинному обучению. В Node2Vec модифицирован способ генерации самих случайных блужданий.

В настоящей главе мы поговорим об этих модификациях и о том, как найти лучшие параметры для данного графа. Мы реализуем архитектуру Node2Vec и сравним ее с использованием DeepWalk на примере Клуба карате Закари. Это даст вам хорошее понимание различий между двумя архитектурами. Наконец, мы воспользуемся этой технологией для решения реальной задачи – создания рекомендательной системы фильмов (recommender system – RecSys) на базе Node2Vec.

К концу этой главы вы узнаете, как обучить Node2Vec на любом наборе графовых данных и как подобрать подходящие параметры. Вы поймете, почему эта архитектура в целом работает лучше, чем DeepWalk, и как ее применять для решения задач.

В этой главе будут рассмотрены следующие темы:

- «Знакомство с Node2Vec»;
- «Реализация Node2Vec»;
- «Создание рекомендательной системы фильмов».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter04.

Знакомство с Node2Vec

Node2Vec был представлен в 2016 году Гровером и Лесковцем из Стэнфордского университета [1]. Он сохраняет те же два основных компонента, что и DeepWalk: случайные блуждания и Word2Vec. Разница в том, что вместо получения последовательностей узлов с равномерным распределением в Node2Vec мы получаем смещенные случайные блуждания. В двух следующих разделах мы увидим, почему эти **смещенные случайные блуждания** (biased random walks) работают лучше и как их реализовать:

- «Определение окрестности (neighborhood)»;
- «Внесение смещений в случайные блуждания».

Давайте начнем с того, что поставим под сомнение нашу интуитивную концепцию окрестностей.

Определение окрестности

Как определить окрестность узла? Ключевая концепция, представленная в Node2Vec, – это гибкое понятие окрестности. Интуитивно мы думаем об этом как о чем-то близко расположенном к исходному узлу, но что означает «близко» в контексте графа? В качестве примера возьмем следующий граф:

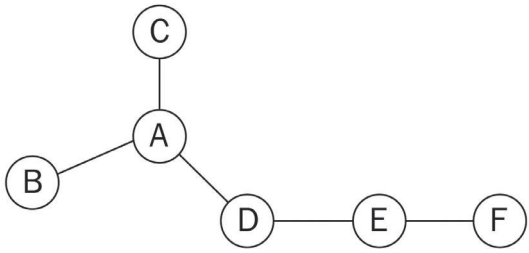


Рис. 4.1 ❖ Пример случайного графа

Мы хотим исследовать три узла в окрестностях узла **A**. Этот процесс исследования также называется **стратегией отбора** (sampling strategy).

- Возможное решение может состоять в том, чтобы рассмотреть три ближайших узла с точки зрения соединений. В таком случае окрестностью узла A, обозначаемой как $N(A)$, будет $N(A) = \{B, C, D\}$.

- Другая возможная стратегия отбора состоит в том, чтобы сначала выбрать узлы, которые не являются смежными с предыдущими узлами. В нашем примере окрестность узла A будет $N(A) = \{D, E, F\}$.

Другими словами, мы хотим реализовать **поиск в ширину** (Breadth-First Search – BFS) в первом случае и **поиск в глубину** (Depth-First Search – DFS) во втором. Дополнительную информацию об этих алгоритмах и реализациях можно найти в главе 2 «Теория графов для графовых нейронных сетей».

Здесь важно отметить, что эти стратегии отбора демонстрируют по отношению друг к другу противоположное поведение: BFS сфокусирован на локальной сети вокруг узла, в то время как DFS в большей степени создает макропредставление графа. Учитывая наше интуитивное определение окрестности, возникает соблазн просто отказаться от DFS. Однако авторы Node2Vec утверждают, что это было бы ошибкой: каждый подход фиксирует разное, но при этом ценное представление сети. Они считают, что между этими алгоритмами и двумя свойствами сети существет связь:

- **структурная эквивалентность** (structural equivalence). Это означает, что узлы структурно эквивалентны, если у них много общих соседей. Таким образом, если у них много общих соседей, их структурная эквивалентность выше;
- **гомофилия** (homophily), как было замечено ранее, предполагает, что схожие узлы с большей вероятностью будут соединены.

Авторы Node2Vec утверждают, что BFS идеально подходит для выявления структурной эквивалентности, поскольку эта стратегия рассматривает только соседние узлы. В этих случайных блужданиях узлы часто повторяются и остаются близко расположенными друг к другу. В DFS отобранные узлы точнее отражают макровзгляд на окрестность, что важно для выявления сообществ на основе гомофилии. Однако при уходе на большую глубину отбираются узлы, которые находятся далеко от исходного и, таким образом, становятся менее репрезентативными. Вот почему мы ищем компромисс между этими двумя свойствами: гомофилия может быть более полезна для понимания определенных графов, и наоборот.

Если вас смущает эта связь, вы не одиноки: авторы некоторых статей и блогов ошибочно предполагают, что BFS помогает выявить гомофилию, а DFS связан со структурной эквивалентностью. В любом случае мы считаем искомым решением графы, сочетающие гомофилию и структурную эквивалентность. Вот почему, независимо от этих связей, мы хотим использовать обе стратегии отбора для создания нашего набора данных.

Давайте посмотрим, как мы можем реализовать их для генерации случайных блужданий.

Внесение смещений в случайные блуждания

Напомним, что случайные блуждания – это последовательности узлов, случайно выбранных в графе. У них есть начальная точка, которая также может

быть случайной, и заранее заданная длина. Узлы, которые часто появляются вместе в этих блужданиях, подобны словам, которые появляются вместе в предложениях: согласно гипотезе о гомофилии, они имеют схожий смысл и, следовательно, одинаковое представление.

В Node2Vec наша цель – сместить случайность этих блужданий в соответствии с одной из двух стратегий:

- отдаем приоритет узлам, не связанным с предыдущим узлом (аналогично DFS);
- отдаем приоритет узлам, близким к предыдущему узлу (аналогично BFS).

В качестве примера возьмем рис. 4.2. Текущий узел назовем j, предыдущий узел – i, а будущий узел – k. Пусть π_{jk} – ненормализованная вероятность перехода из узла j в узел k. Эту вероятность можно расписать как $\pi_{jk} = \alpha(i, k) \cdot \omega_{jk}$, где $\alpha(i, k)$ – смещение поиска (search bias) между узлами i и k, а ω_{jk} – вес ребра от j до k.

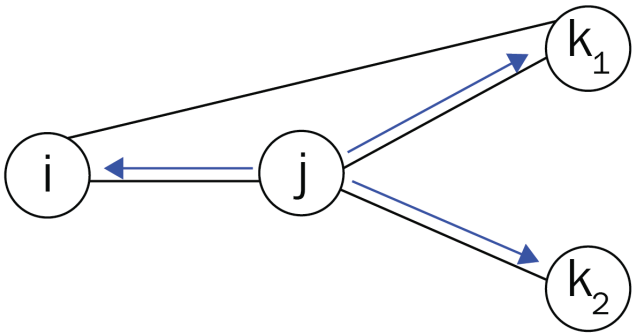


Рис. 4.2 ❖ Пример случайного графа

В DeepWalk у нас $\alpha(a, b) = 1$ для любой пары узлов a и b. В Node2Vec значение $\alpha(a, b)$ определяется на основе расстояния между узлами и двух дополнительных параметров: p – параметра return и q – параметра in-out. Их роль заключается в аппроксимации DFS и BFS соответственно.

Вот как определяется значение $\alpha(a, b)$:

$$\alpha(a,b) = \begin{cases} \frac{1}{p}, & \text{если } d_{ab} = 0 \\ 1, & \text{если } d_{ab} = 1. \\ \frac{1}{q}, & \text{если } d_{ab} = 2 \end{cases}$$

Здесь d_{ab} – это кратчайшее расстояние между узлами a и b. Мы можем обновить ненормализованную вероятность перехода в предыдущем графе следующим образом:

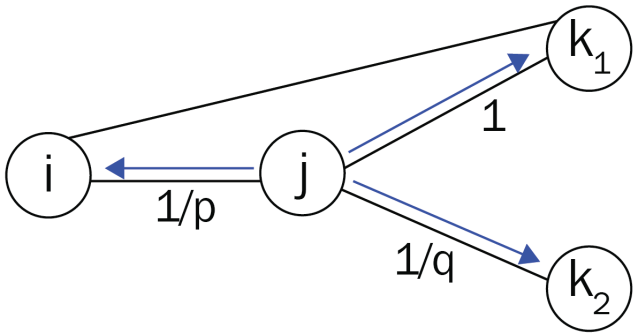


Рис. 4.3 ❖ Граф с вероятностями перехода

Поясним эти вероятности.

- Случайное блуждание начинается с узла i и теперь достигает узла j. Вероятность возврата к предыдущему узлу i контролируется параметром p. Чем выше значение параметра p, тем меньше вероятность возврата к предыдущему узлу, т. е. тем больше случайное блуждание будет исследовать новые узлы вместо повторного посещения уже посещенных и будет выглядеть как DFS.
- Ненормализованная вероятность перехода к узлу k_1 равна 1, поскольку этот узел находится в непосредственной близости от нашего предыдущего узла i.
- Наконец, вероятность перехода к узлу k_2 контролируется параметром q. Чем выше значение параметра q, тем больше случайное блуждание будет фокусироваться на узлах, близких к предыдущему, и будет выглядеть как BFS.

Лучший способ понять вышесказанное – реализовать эту архитектуру и поэкспериментировать с параметрами. Давайте сделаем это пошагово на примере случайного графа, взятого из предыдущей главы и показанного на рис. 4.4.

```
import networkx as nx
import matplotlib.pyplot as plt

# создаем граф
G = nx.erdos_renyi_graph(10, 0.3, seed=1, directed=False)

# визуализируем граф
plt.figure()
plt.axis('off')
nx.draw_networkx(G,
                 pos=nx.spring_layout(G, seed=0),
                 node_size=600,
                 cmap='coolwarm',
                 font_size=14,
                 font_color='white'
                 )
```

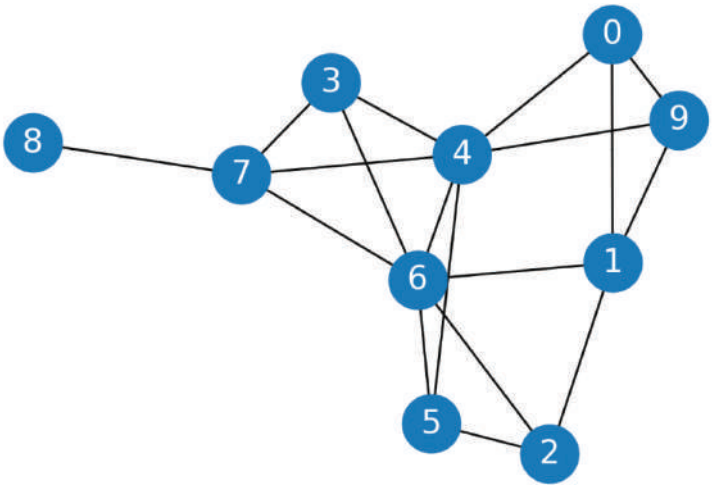


Рис. 4.4 ❖ Случайный граф

Обратите внимание, что это невзвешенный граф, поэтому вероятность перехода определяется только смещением поиска, вес ребра у нас всегда будет равен 1.

Во-первых, мы хотим написать функцию, которая будет случайным образом выбирать следующий узел в графе на основе предыдущего узла, текущего узла и двух параметров p и q.

1. Начнем с импорта необходимой библиотеки numpy:

```
import numpy as np
```

2. Мы задаем функцию `next_node()` со списком наших параметров:

```
def next_node(previous, current, p, q):
```

3. Извлекаем список узлов, являющихся соседями текущего узла, и инициализируем список значений α:

```
    # создаем пустой список для значений alpha
    alphas = []

    # получаем список узлов, являющихся
    # соседями текущего узла
    neighbors = list(G.neighbors(current))
```

4. Для каждого соседа нам нужно вычислить соответствующее значение α: $\frac{1}{p}$, если этот сосед – предыдущий узел, 1, если этот сосед соединен с предыдущим узлом, и $\frac{1}{q}$ в противном случае:

```
    # вычисляем соответствующее значение alpha для каждого соседа
    for neighbor in neighbors:
        # расстояние = 0: вероятность возврата к предыдущему узлу
```

```
        if neighbor == previous:
            alpha = 1/p
        # расстояние = 1: вероятность посещения локального узла
        elif G.has_edge(neighbor, previous):
            alpha = 1
        # расстояние = 2: вероятность исследовать неизвестный узел
        else:
            alpha = 1/q
        alphas.append(alpha)
```

5. Теперь нормализуем значения α, чтобы получить вероятности:

```
    # нормализуем значения alpha, чтобы получить вероятности перехода
    probs = [alpha / sum(alphas) for alpha in alphas]
```

6. Мы случайным образом выбираем следующий узел на основе вероятностей перехода, рассчитанных на предыдущем шаге с помощью функции `np.random.choice()`, и возвращаем его:

```
    # случайно выбираем новый узел на основе вероятностей перехода
    next = np.random.choice(neighbors, size=1, p=probs)[0]
    return next
```

Теперь пишем новую версию функции `random_walk()` для генерации случайных блужданий, в рамкой которой мы и протестируем ранее написанную функцию `next_node()`.

Способ генерации этих случайных блужданий будет аналогичен тому, что мы видели в предыдущей главе. Разница состоит лишь в том, что следующий узел выбирается с помощью функции `next_node()`, которая требует дополнительных параметров: p и q, а также предыдущий и текущий узлы. Эти узлы можно легко получить, просмотрев два последних элемента, добавленных в список `walk`. Кроме того, мы возвращаем строки вместо целых чисел по соображениям совместимости.

```
def random_walk(start, length, p, q):
    walk = [start]

    for i in range(length):
        current = walk[-1]
        previous = walk[-2] if len(walk) > 1 else None
        next = next_node(previous, current, p, q)
        walk.append(next)

    return walk
```

Теперь у нас есть все необходимое для генерации случайных блужданий. Давайте сгенерируем случайное блуждание длиной 5 с $p = 1$ и $q = 1$.

```
np.random.seed(0)

random_walk(0, 8, p=1, q=1)
```

Функция `random_walk()` создает следующую последовательность:

```
[0, 4, 7, 6, 4, 5, 4, 5, 6]
```

Процесс должен быть случайным, поскольку каждый соседний узел имеет одинаковую вероятность перехода. С этими параметрами мы воспроизводим точный алгоритм DeepWalk.

Теперь давайте увеличим вероятность возвращения к предыдущему узлу, задав $q = 10$.

```
np.random.seed(0)

random_walk(0, 8, p=1, q=10)

[0, 4, 9, 4, 7, 4, 6, 4, 7]
```

Мы видим, что последовательность содержит большое количество повторов одного и того же узла (узлы 4 и 7).

Теперь давайте увеличим вероятность исследования новых узлов, задав $p = 10$.

```
np.random.seed(0)

random_walk(0, 8, p=10, q=1)

0, 4, 7, 6, 3, 4, 7, 6, 5]
```

На этот раз исследовано большее количество узлов в графе.

Давайте посмотрим, как использовать эти свойства на реальном примере, и сравним результаты с DeepWalk.

Реализация Node2Vec

Теперь, когда у нас есть функции для генерации смещенных случайных блужданий, реализация Node2Vec становится очень похожей на реализацию DeepWalk. Она настолько похожа, что мы можем повторно воспользоваться одним и тем же программным кодом и создавать последовательности с $p = 1$ и $q = 1$ для реализации DeepWalk как частного случая Node2Vec. Давайте повторно воспользуемся Клубом карате Закари для этой задачи.

Как и в предыдущей главе, наша цель – правильно отнести каждого участника клуба к одной из двух групп (`'Mr. Hi'` и `'Officer'`). Мы будем использовать эмбеддинги узлов, полученные с помощью Node2Vec, в качестве входных данных для классификатора – модели машинного обучения (в данном случае используем случайный лес).

Давайте посмотрим, как это реализовать шаг за шагом.

1. Во-первых, убедитесь, что у вас установлена библиотека gensim для использования Word2Vec:

```
!pip install -qI gensim==4.3.0
```

2. Импортируем необходимые классы и функции:

```
from gensim.models.word2vec import Word2Vec
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
```

3. Загружаем набор данных Клуб карате Закари:

```
# загружаем набор данных
G = nx.karate_club_graph()
```

4. Преобразовываем метки узлов в целочисленные значения (`0` и `1`):

```
# создаем метки (Mr. Hi = 0, Officer = 1)
labels = []
for node in G.nodes:
    label = G.nodes[node]['club']
    labels.append(1 if label == 'Officer' else 0)
```

5. Мы генерируем список случайных блужданий, используя нашу функцию `random_walk()` 80 раз для каждого узла графа. Параметры *p* и *q* равны 3 и 2 соответственно:

```
# создаем список случайных блужданий
walks = []
for node in G.nodes:
    for _ in range(80):
        walks.append(random_walk(node, 10, 3, 2))
```

6. Создаем экземпляр класса `Word2Vec`. Здесь мы инициализируем модель skip-gram с иерархической функцией softmax:

```
# создаем и обучаем Word2Vec для DeepWalk
node2vec = Word2Vec(walks,
                    hs=1, # иерархический softmax
                    sg=1, # skip-gram
                    vector_size=100,
                    window=10,
                    workers=2,
                    min_count=1,
                    seed=0)
```

7. Затем просто обучаем модель на сгенерированных нами случайных блужданиях, используя 30 эпох:

```
node2vec.train(walks,
               total_examples=node2vec.corpus_count,
               epochs=30,
               report_delay=1)
```

8. Создаем маски для обучения и тестирования классификатора.

```
# создаем маски для обучения и тестирования модели
train_mask = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
test_mask = [0, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21,
             23, 25, 26, 27, 28, 29, 30, 31, 32, 33]
labels = np.array(labels)
```

9. Обучаем классификатор – модель случайного леса на обучающих данных:

```
# обучаем классификатор на данных, полученных с помощью Node2Vec
clf = RandomForestClassifier(random_state=0)
clf.fit(node2vec.wv[train_mask], labels[train_mask])
```

10. Теперь оцениваем качество на тестовых данных с точки зрения правильности:

```
# оцениваем качество модели
y_pred = clf.predict(node2vec.wv[test_mask])
acc = accuracy_score(y_pred, labels[test_mask])
print(f'Правильность на данных Node2Vec = {acc*100:.2f}%')
```

```
Правильность на данных Node2Vec = 90.91%
```

Чтобы реализовать DeepWalk, мы можем в точности повторить ту же самую процедуру с $p = 1$ и $q = 1$. Однако, чтобы провести справедливое сравнение, мы не можем использовать точечную оценку правильности. Действительно, здесь задействовано множество стохастических процессов – может не повезти, и мы получим лучший результат от наихудшей модели.

Чтобы ограничить случайность наших результатов, мы можем повторить этот процесс 10 раз (рекомендуется 100 раз) и взять среднее значение. Этот результат намного более стабилен и может даже включать стандартное отклонение (можно воспользоваться функцией `np.std()`) для измерения изменчивости правильности.

Но, прежде чем мы это сделаем, давайте поиграем в игру. В предыдущей главе мы говорили о Клубе карате Закари как о гомофильной сети. Это свойство выявляет DFS (поиск в глубину), увеличение параметра p способствует поиску в глубину. Если это утверждение и связь между DFS и гомофилией верны, мы получим более высокую правильность с более высокими значениями p.

Я повторил тот же эксперимент для значений p и q от 1 до 7. В реальном проекте машинного обучения мы воспользовались бы проверочными данными при выполнении поиска оптимальных значений параметров p и q. В этом примере мы используем тестовые данные, поскольку это исследование уже является нашим итоговым решением.

```
import pandas as pd

# создаем списки для хранения комбинаций p и q,
# средних значений правильности, стандартных
# отклонений правильности
```

```
combinations = []
mean_acc_scores = []
std_acc_scores = []

for p in [1, 2, 3, 4, 5, 6, 7]:
    for q in [1, 2, 3, 4, 5, 6, 7]:

        # записываем комбинации p и q
        combinations.append(f"p={p} и q={q}")

        # создаем пустой лист для хранения значений правильности
        acc_lst = []

        # 10 раз повторяем следующие шаги
        for _ in range(10):
            # создаем список случайных блужданий
            walks = []
            for node in G.nodes:
                for _ in range(80):
                    walks.append(random_walk(node, 10, p, q))

            # создаем и обучаем Word2Vec для DeepWalk
            node2vec = Word2Vec(walks,
                                hs=1, # иерархический softmax
                                sg=1, # skip-gram
                                vector_size=100,
                                window=10,
                                workers=2,
                                min_count=1,
                                seed=0)

            node2vec.train(walks,
                           total_examples=node2vec.corpus_count,
                           epochs=30,
                           report_delay=1)

            # обучаем классификатор на данных, полученных с помощью Node2Vec
            clf = RandomForestClassifier(random_state=0)
            clf.fit(node2vec.wv[train_mask], labels[train_mask])

            # оцениваем качество модели
            y_pred = clf.predict(node2vec.wv[test_mask])
            acc = accuracy_score(y_pred, labels[test_mask])
            # добавляем значение правильности в список
            acc_lst.append(acc)

        # расчет среднего значения правильности
        acc_mean = np.mean(acc_lst)
        # расчет стандартного отклонения правильности
        acc_std = np.std(acc_lst)

        # добавляем средние значения и стандартные отклонения
        # в соответствующие списки
        mean_acc_scores.append(acc_mean * 100)
```

```
        std_acc_scores.append(acc_std * 100)

# вычисляем средние значения правильности со штрафом
penalty_mean_acc_scores = np.subtract(
    np.array(mean_acc_scores), np.array(std_acc_scores)
)

# создаем датафрейм
df = pd.DataFrame({"Правильность mean": mean_acc_scores,
                   "Правильность std": std_acc_scores,
                   "Правильность со штрафом": penalty_mean_acc_scores},
                   index=combinations)

# сортируем по усредненной правильности со штрафом
df = df.sort_values(by='Правильность со штрафом', ascending=False)
df
```

	Правильность mean	Правильность std	Правильность со штрафом
p=5 и q=4	96.818182	2.082989	94.735193
p=4 и q=3	97.272727	3.015113	94.257614
p=5 и q=3	96.363636	2.727273	93.636364
p=6 и q=4	96.363636	2.727273	93.636364
p=5 и q=6	96.363636	2.727273	93.636364
p=7 и q=7	95.909091	2.447802	93.461289
p=6 и q=2	96.818182	3.550113	93.268068
p=3 и q=4	96.363636	3.401507	92.962130
p=6 и q=7	96.363636	3.401507	92.962130
p=4 и q=2	96.363636	3.401507	92.962130
p=6 и q=6	94.545455	1.818182	92.727273
p=7 и q=6	95.909091	3.181818	92.727273
p=5 и q=7	95.909091	3.181818	92.727273
p=2 и q=2	95.909091	3.181818	92.727273
p=6 и q=5	95.909091	3.181818	92.727273
p=1 и q=6	95.454545	2.874798	92.579748
p=3 и q=2	95.454545	2.874798	92.579748
p=3 и q=3	95.454545	2.874798	92.579748
p=1 и q=2	95.454545	2.874798	92.579748
p=7 и q=4	96.363636	3.962635	92.401001
p=6 и q=1	96.818182	4.568125	92.250057
p=3 и q=6	94.090909	2.082989	92.007920
p=5 и q=2	96.363636	4.453618	91.910019

В целях экономии места таблица приведена не полностью. Здесь у нас несколько примечательных результатов.

- Для этого набора DeepWalk ($p = 1$ и $q = 1$) работает хуже, чем любая другая комбинация, описанная здесь, и данный эксперимент показывает, насколько полезными могут быть смещенные случайные блуждания. Однако это не всегда так: несмещенные случайные блуждания также могут лучше работать с другими наборами данных.
- Высокие значения p приводят к повышению качества, что подтверждает нашу гипотезу. Знание того, что перед нами – социальная сеть, убедительно свидетельствует о том, что смещение наших случайных блужданий в сторону гомофилии является хорошей стратегией. Об этом следует помнить, работая графами, представляющими социальные сети.

Не стесняйтесь экспериментировать с параметрами и пытаться найти другие интересные результаты. Мы могли бы исследовать результаты с очень высокими значениями p (> 7) или, наоборот, значениями p в диапазоне от 0 до 1.

Клуб карате Закари – это базовый набор данных, но в следующем разделе мы увидим, как можно использовать метод Node2Vec для решения гораздо более интересных задач.

Создание рекомендательной системы фильмов

Один из самых популярных примеров применения GNN – это рекомендательные системы. Если задуматься об основе Word2Vec (и, следовательно, DeepWalk и Node2Vec), то цель состоит в том, чтобы создавать векторы с возможностью измерения их сходства. Кодируйте фильмы вместо слов, и вы внезапно сможете получить фильмы, наиболее похожие на кинокартины, заданные нами на входе. Это очень похоже на рекомендательную систему, не так ли?

Но как закодировать фильмы? Мы хотим создать (смещенные) случайные блуждания фильмов, но для этого требуется набор графовых данных, в котором похожие друга на друга фильмы связаны друг с другом. Подобный поиск не будет простым.

Другой подход – посмотреть на рейтинги пользователей. Существуют разные методы построения графа на основе рейтингов: двудольные графы, ребра на основе поточечной взаимной информации и т. д. В этом разделе мы реализуем простой и интуитивно понятный подход: соединяем фильмы, которые нравятся одним и тем же пользователям. Затем на основе этого графа мы с помощью Node2Vec получим эмбеддинги фильмов.

1. Сначала давайте загрузим набор данных. MovieLens [2] – популярный выбор: небольшая версия последнего набора данных (09/2018) включает 100 836 рейтингов, 9742 фильма и 610 пользователей. Мы можем скачать его с помощью следующего программного кода Python:

```
from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile

url = 'https://files.grouplens.org/datasets/movielens/ml-100k.zip'
with urlopen(url) as zurl:
    with ZipFile(BytesIO(zurl.read())) as zfile:
        zfile.extractall('.')
```

2. Нас интересуют два файла в папке `ml-100k` – `u.data` и `u.item`. В первом хранятся все рейтинги пользователей, а второй позволяет перевести идентификаторы фильмов в названия.
3. Давайте посмотрим, как выглядят рейтинги, прочитав файл `u.data` в датафрейм pandas с помощью функции `pd.read_csv()`:

```
ratings = pd.read_csv('ml-100k/u.data', sep='\t',
                      names=['user_id', 'movie_id',
                             'rating', 'unix_timestamp'])
ratings
```

	user_id	movie_id	rating	unix_timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596
...	...	...	...	...
99995	880	476	3	880175444
99996	716	204	5	879795543
99997	276	1090	1	874795795
99998	13	225	2	882399156
99999	12	203	3	879959583

100000 rows × 4 columns

4. Теперь прочитаем файл `u.item`:

```
movies = pd.read_csv('ml-100k/u.item', sep='|',
                     usecols=range(2),
                     names=['movie_id', 'title'],
                     encoding='latin-1')
movies
```

	movie_id	title
0	1	Toy Story (1995)
1	2	GoldenEye (1995)
2	3	Four Rooms (1995)
3	4	Get Shorty (1995)
4	5	Copycat (1995)
...	...	...
1677	1678	Mat' i syn (1997)
1678	1679	B. Monkey (1998)
1679	1680	Sliding Doors (1998)
1680	1681	You So Crazy (1994)
1681	1682	Scream of Stone (Schrei aus Stein) (1991)

1682 rows × 2 columns

5. В данном случае нам нужны фильмы, которые понравились одним и тем же пользователям. Это означает, что такие рейтинги, как 1, 2 и 3, не очень релевантны. Мы можем отбросить их и оставить фильмы только с рейтингами 4 и 5:

```
# оставляем фильмы только с высокими рейтингами
ratings = ratings[ratings.rating >= 4]
ratings
```

	user_id	movie_id	rating	unix_timestamp
5	298	474	4	884182806
7	253	465	5	891628467
11	286	1014	5	879781125
12	200	222	5	876042340
16	122	387	5	879270459
...	...	...	...	...
99988	421	498	4	892241344
99989	495	1091	4	888637503
99990	806	421	4	882388897
99991	676	538	4	892685437
99996	716	204	5	879795543

55375 rows × 4 columns

6. Сейчас у нас 55 375 рейтингов от 942 пользователей. Следующий шаг – подсчитать, сколько раз одному и тому же пользователю понравились два фильма. Мы получим такую оценку для каждого пользователя в наборе данных.
7. Чтобы упростить задачу, воспользуемся структурой данных `defaultdict`, которая автоматически создает пропущенные записи, а не выдает ошибку. Мы будем использовать эту структуру для подсчета пар фильмов, которые нравятся пользователям:

```
from collections import defaultdict
pairs = defaultdict(int)
```

8. Теперь по каждому пользователю в списке пользователей в нашем наборе мы получаем список фильмов, которые понравились текущему пользователю, и увеличиваем счетчик, заданный для пары фильмов, каждый раз, когда эти фильмы оказываются вместе в одном списке:

```
# по каждому пользователю в списке пользователей
for group in ratings.groupby("user_id"):
    # список идентификаторов фильмов, которые
    # понравились текущему пользователю
    user_movies = list(group[1]["movie_id"])

    # подсчитываем, сколько раз два фильма оказались
    # в одном списке
    for i in range(len(user_movies)):
        for j in range(i+1, len(user_movies)):
            pairs[(user_movies[i], user_movies[j])] += 1
```

9. Объект `pairs` теперь хранит информацию о том, сколько раз два фильма понравились одному и тому же пользователю. Мы можем использовать эту информацию для построения ребер нашего графа.
10. Создаем граф с помощью библиотеки networkx:

```
# создаем граф
G = nx.Graph()
```

11. Для каждой пары фильмов в нашем объекте `pairs` мы извлекаем два фильма и соответствующую им оценку (значение счетчика):

```
# пытаемся создать ребро между двумя фильмами,
# которые оба понравились пользователю
for pair in pairs:
    movie1, movie2 = pair
    score = pairs[pair]
```

12. Если эта оценка выше 10, мы создаем в графе взвешенное ребро, чтобы соединить оба фильма на основе этой оценки. Мы не рассматриваем оценки ниже 10, потому что это даст нам большой граф, в котором связи будут менее значимыми:

```
            # ребро создаем только тогда, когда оценка
            # является достаточно высокой
            if score >= 20:
                G.add_edge(movie1, movie2, weight=score)
```

Созданный нами граф состоит из 410 узлов (фильмов) и 14 936 ребер:

```
print("Общее количество узлов в графе:", G.number_of_nodes())
print("Общее количество ребер в графе:", G.number_of_edges())
```

```
Общее количество узлов в графе: 410
Общее количество ребер в графе: 14936
```

Теперь можно обучить на нем Node2Vec для получения эмбеддингов узлов!

Мы могли бы повторно воспользоваться нашей реализацией из предыдущего раздела, но на самом деле существует целая библиотека Python, посвященная Node2Vec (также под названием node2vec). Ее можно установить с помощью команды:

```
!pip install -q node2vec==0.4.6
```

1. Давайте импортируем класс `Node2Vec` библиотеки node2vec:

   ```
   from node2vec import Node2Vec
   ```

2. Мы создаем экземпляр класса `Node2Vec`, который будет автоматически генерировать смещенные случайные блуждания на основе параметров p и q:

   ```
   from node2vec import Node2Vec

   node2vec = Node2Vec(G, dimensions=64, walk_length=20,
                       num_walks=200, p=2, q=1, workers=1)
   ```

3. Мы обучаем модель, используя смещенные случайные блуждания и задав значение параметра `window` равным 5:

   ```
   model = node2vec.fit(window=10, min_count=1, batch_words=4)
   ```

   Параметр `window` указывает на размер окна при создании случайных блужданий по графу. Мы уже знаем, что, когда Node2Vec используется для получения эмбеддингов узлов, он выполняет случайные блуждания по графу, чтобы собрать информацию о соседях каждого узла. Размер окна `window` определяет максимальное расстояние между текущим узлом и его соседями при выполнении блужданий. Например, если `window=5`, то при блуждании будут учтены 5 узлов до текущего узла и 5 узлов после текущего узла. Это позволяет алгоритму учитывать контекст соседей во время обучения.

4. Давайте напишем функцию `recommend()`, которая рекомендует фильмы на основе введенного названия. Она принимает на вход название фильма. Далее происходит преобразование названия в идентифика-

тор фильма, который мы можем использовать для обращения к нашей модели:

```
def recommend(movie):
    movie_id = str(movies[movies.title == movie].movie_id.values[0])
```

5. Мы перебираем пять наиболее похожих векторов слов. Преобразуем эти идентификаторы в названия фильмов, которые печатаем с соответствующими значениями сходства:

```
    for id in model.wv.most_similar(movie_id)[:5]:
        title = movies[movies.movie_id == int(id[0])].title.values[0]
        print(f'{title}: {id[1]:.2f}')
```

6. Мы вызываем функцию `recommend()`, чтобы получить пять фильмов, которые наиболее похожи на «Star Wars» («Звездные войны») с точки зрения косинусного сходства:

```
recommend('Star Wars (1977)')
```

```
Return of the Jedi (1983): 0.58
Raiders of the Lost Ark (1981): 0.53
Monty Python and the Holy Grail (1974): 0.48
Indiana Jones and the Last Crusade (1989): 0.47
Amadeus (1984): 0.46
```

Модель говорит нам, что «Return of the Jedi» («Возвращение джедая») и «Raiders of the Lost Ark» («В поисках утраченного ковчега») наиболее похожи на «Звездные войны», хотя и с относительно низким значением сходства (< 0.7). Тем не менее это хороший результат для нашего первого шага в мир рекомендательных систем! В последующих главах мы увидим более мощные модели и подходы к созданию современной рекомендательной системы.

Выводы

В этой главе мы узнали о Node2Vec, второй популярной архитектуре для извлечения эмбеддингов. Мы реализовали функции для генерации смещенных случайных блужданий и объяснили связь между их параметрами и двумя свойствами сети: гомофилией и структурной эквивалентностью. Мы показали их полезность, сравнив результаты Node2Vec с результатами DeepWalk для Клуба карате Закари. Наконец, создали нашу первую рекомендательную систему, используя собственный набор графовых данных и альтернативную реализацию Node2Vec. Она дала правильные рекомендации, которые мы еще улучшим в последующих главах.

В главе 5 «Включение информации о характеристиках узлов с помощью простых нейронных сетей» мы поговорим об одной упущенной из виду проблеме, касающейся DeepWalk и Node2Vec: отсутствии информации о харак-

теристиках узлов. Мы попытаемся решить эту проблему с помощью простых нейронных сетей, которые не способны распознать топологию сети. Эту дилемму важно понять, прежде чем мы в итоге перейдем к графовым нейронным сетям.

Дополнительное чтение

[1] A. Grover and J. Leskovec, node2vec: Scalable Feature Learning for Networks. arXiv, 2016. DOI:10.48550/ARXIV.1607.00653. Доступ по ссылке https://arxiv.org/abs/1607.00653.

[2] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. https://doi.org/10.1145/2827872. Доступ по ссылке https://dl.acm.org/doi/10.1145/2827872.

Глава 5

Включение информации о характеристиках узлов с помощью простых нейронных сетей

До сих пор единственным типом информации, который мы рассматривали, была топология графа. Однако графовые наборы данных, как правило, богаче, чем просто набор связей: узлы и ребра могут обладать характеристиками (признаками), представляющими оценки, цвета, слои и т. д. Включение этой дополнительной информации в наши входные данные позволяет получить эмбеддинги лучшего качества. На самом деле это довольно естественно для машинного обучения: узлы и ребра имеют ту же структуру, что и табличный (не графовый) набор данных. Это означает, что к этим данным можно применять традиционные методы, например нейронные сети.

В этой главе мы представим два новых графовых набора данных: `Cora` и `Facebook Page-Page`. Мы увидим, как простые нейронные сети (Vanilla Neural Networks) работают с характеристиками узла, рассматривая их только как табличные наборы данных. Затем поэкспериментируем, чтобы включить в наши нейронные сети топологическую информацию. Так мы получим нашу первую архитектуру GNN: простую модель, которая учитывает как характеристики узла, так и ребра между узлами. Наконец, мы сравним качество прогнозирования при использовании этих двух архитектур и получим один из наиболее важных результатов этой книги.

К концу этой главы вы освоите реализацию простых нейронных сетей и простых GNN в PyTorch. Вы сможете встраивать топологические признаки в представления узлов, что является основой любой архитектуры GNN. Это позволит вам значительно повысить качество ваших моделей за счет преобразования табличных наборов данных в задачи на графах.

В этой главе будут рассмотрены следующие темы:

- «Знакомство с графовыми наборами данных»,
- «Классификация узлов с помощью простых нейронных сетей»,
- «Классификация узлов с помощью простых графовых нейронных сетей».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter05.

Знакомство с графовыми наборами данных

Графовые наборы данных, которые мы собираемся использовать в этой главе, являются более сложными, чем Клуб карате Закари: они имеют больше узлов, больше ребер и включают признаки. В этом разделе мы познакомимся с ними, чтобы лучше понять эти графы и выбрать способы их обработки с помощью PyTorch Geometric. Вот два набора данных, которые мы будем использовать:

- набор данных `Cora`,
- набор данных `Facebook Page-Page`.

Начнем с меньшего по размеру набора – популярного набора данных `Cora`.

Набор данных Cora

Представленный Sen et al. в 2008 году [1], набор данных `Cora` (без лицензии) стал самым популярным набором данных для классификации узлов в научной литературе. Он представляет собой сеть, состоящую из 2708 публикаций. Ребра в графе представляют связи цитирования между публикациями. Каждая публикация описана в виде бинарного вектора из 1433 уникальных слов, где 0 и 1 обозначают отсутствие или наличие соответствующего слова. В обработке естественного языка это представление еще называется **мешком слов** (bag of words). Каждой публикации присвоены метки (классы), обозначающие тему публикации. Наша цель – отнести каждый узел к одному из семи классов.

Независимо от типа данных визуализация всегда является важным шагом на пути к лучшему пониманию решаемой задачи. Однако графы могут быстро

стать слишком большими для визуализации с помощью таких Python'овских библиотек, как networkx. Вот почему специально для визуализации графовых данных были разработаны специальные инструменты. В этой книге мы используем два наиболее популярных из них: **yEd Live** (https://www.yworks.com/yed-live/) и **Gephi** (https://gephi.org/).

На рис. 5.1 показан граф, иллюстрирующий набор данных Cora и созданный с помощью yEd Live. Вы видите, что узлы, соответствующие публикациям, обозначены оранжевым цветом, а связи между ними – зеленым. Некоторые публикации настолько взаимосвязаны, что образуют кластеры. Скорее всего, эти кластеры будет легче классифицировать, чем плохо связанные узлы.

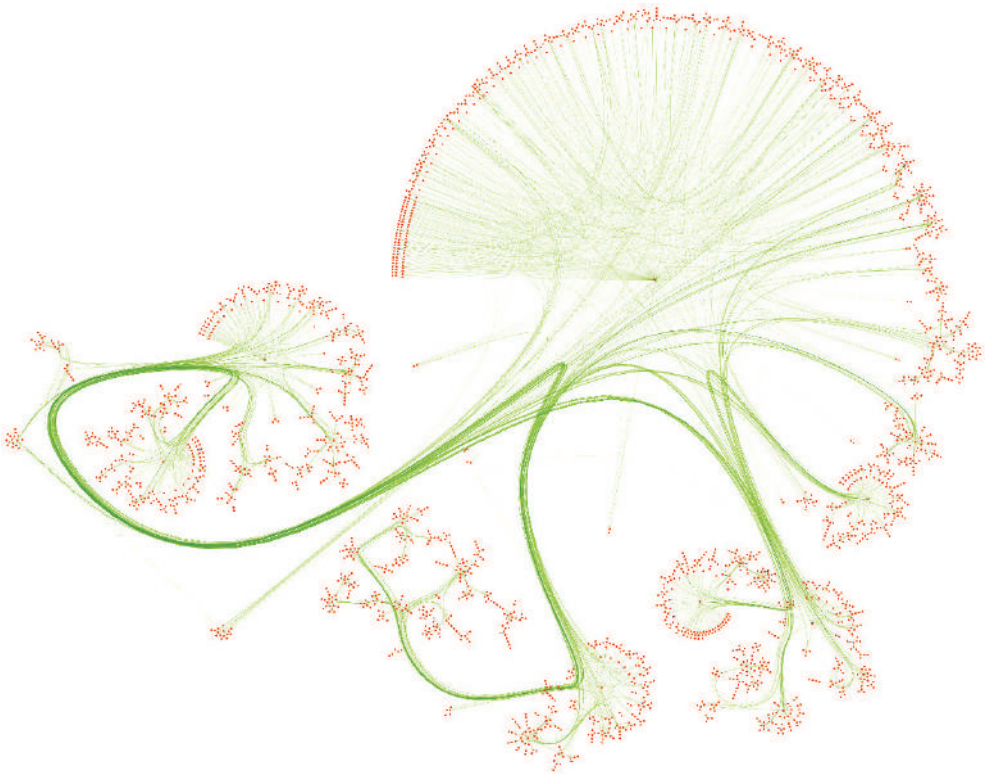


Рис. 5.1 ❖ Набор данных Cora, визуализированный с помощью yEd Live

Давайте загрузим набор данных `Cora` и проанализируем его основные характеристики с помощью PyTorch Geometric. В этой библиотеке есть специальный класс для загрузки этого набора данных и получения информации о графе. Здесь мы предполагаем, что PyTorch Geometric уже установлен.

```
import torch
torch.manual_seed(0)
```

```
torch.cuda.manual_seed(0)
torch.cuda.manual_seed_all(0)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

1. Мы импортируем класс `Planetoid` из PyTorch Geometric:

   ```
   from torch_geometric.datasets import Planetoid
   ```

2. Скачиваем набор `Cora` с помощью этого класса:

   ```
   # импортируем набор из PyTorch Geometric
   dataset = Planetoid(root=".", name="Cora")
   ```

3. У набора `Cora` – один граф, который мы будем хранить в переменной `data`:

   ```
   data = dataset[0]
   ```

4. Давайте напечатаем общую информацию о наборе:

   ```
   # печатаем информацию о наборе данных
   print(f'Набор данных: {dataset}')
   print('---------------')
   print(f'Количество графов: {len(dataset)}')
   print(f'Количество узлов: {data.x.shape[0]}')
   print(f'Количество признаков: {dataset.num_features}')
   print(f'Количество классов: {dataset.num_classes}')
   ```

   ```
   Набор данных: Cora()
   ---------------
   Количество графов: 1
   Количество узлов: 2708
   Количество признаков: 1433
   Количество классов: 7
   ```

5. Кроме того, мы можем получить подробную информацию о характеристиках графа с помощью специальных функций PyTorch Geometric:

   ```
   # печатаем информацию о графе
   print(f'\nГраф:')
   print('------')
   print(f'Ребра являются ориентированными: {data.is_directed()}')
   print(f'У графа есть изолированные узлы: {data.has_isolated_nodes()}')
   print(f'У графа есть петли: {data.has_self_loops()}')
   ```

   ```
   Граф:
   ------
   Ребра являются ориентированными: False
   У графа есть изолированные узлы: False
   У графа есть петли: False
   ```

Первый вывод подтверждает информацию о количестве узлов, признаков и классов. Второй вывод дает больше информации о самом графе: ребра

не являются ориентированными, у каждого узла есть соседи, и в графе нет петель. Мы могли бы протестировать и другие характеристики графа, используя служебные функции PyTorch Geometric, но в этом примере мы не узнаем ничего нового.

Теперь, когда мы знаем больше о наборе `Cora`, давайте посмотрим набор, который более адекватно отражает размер реальных социальных сетей: набор данных `Facebook Page-Page`.

Набор данных Facebook Page-Page

Этот набор данных был представлен Rozemberczki et al. в 2019 году [2]. Он был создан с использованием Facebook Graph API в ноябре 2017 года. В этом наборе данных каждый из 22 470 узлов представляет собой официальную страницу Facebook. Страницы связаны друг с другом, когда между ними есть взаимные лайки. Характеристики узла (128-мерные векторы) созданы на основе текстовых описаний, написанных владельцами этих страниц. Наша цель – отнести каждый узел к одному из четырех классов (класс – это тип владельца страницы, речь может идти о странице политика, компании, телешоу и правительственной организации).

Набор данных `Facebook Page-Page` аналогичен предыдущему набору, речь идет о социальной сети, поставлена задача классификации узлов. Однако есть три основных отличия от Cora:

- количество узлов намного больше (2708 против 22 470);
- размерность характеристик узлов резко уменьшилась (с 1433 до 128); цель состоит в том, чтобы классифицировать каждый узел по четырем классам вместо семи (что проще, поскольку вариантов меньше).

На рис. 5.2 представлена визуализация набора данных с использованием Gephi. Во-первых, узлы с небольшим количеством соединений были исключены для улучшения наглядности. Размер остальных узлов зависит от количества их соединений, а их цвет указывает на класс, к которому они принадлежат. Наконец, были применены два алгоритма укладки: Fruchterman-Reingold и ForceAtlas2.

Мы можем импортировать набор данных `Facebook Page-Page` таким же способом, что и набор `Cora`.

1. Импортируем класс `FacebookPagePage` из PyTorch Geometric:

```
from torch_geometric.datasets import FacebookPagePage
```

2. Скачиваем набор `FacebookPagePage` с помощью этого класса:

```
# импортируем набор из PyTorch Geometric
dataset = FacebookPagePage(root=".")
```

3. У набора `FacebookPagePage` – один граф, который мы будем хранить в переменной `data`:

```
data = dataset[0]
```

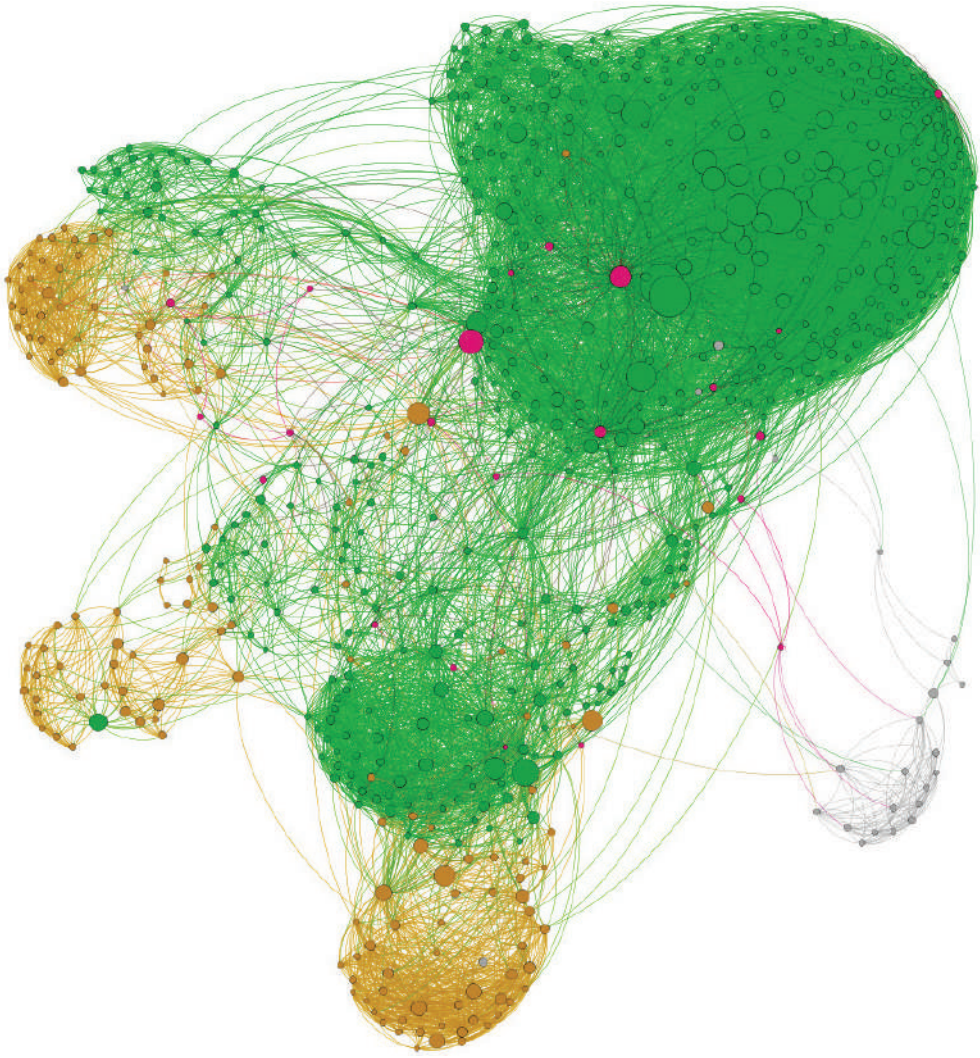


Рис. 5.2 ❖ Набор Facebook Page-Page, визуализированный с помощью Gephi

4. Давайте напечатаем общую информацию о наборе:

```
# печатаем информацию о наборе данных
print(f'Набор данных: {dataset}')
print('---------------')
print(f'Количество графов: {len(dataset)}')
print(f'Количество узлов: {data.x.shape[0]}')
print(f'Количество признаков: {dataset.num_features}')
print(f'Количество классов: {dataset.num_classes}')

Набор данных: FacebookPagePage()
-----------------------
Количество графов: 1
```

```
Количество узлов: 22470
Количество признаков: 128
Количество классов: 4
```

5. Кроме того, мы можем получить подробную информацию о характеристиках графа с помощью специальных функций PyTorch Geometric:

```
# печатаем информацию о графе
print(f'\nГраф:')
print('------')
print(f'Ребра являются ориентированными: {data.is_directed()}')
print(f'У графа есть изолированные узлы: {data.has_isolated_nodes()}')
print(f'У графа есть петли: {data.has_self_loops()}')
```

```
Граф:
------
Ребра являются ориентированными: False
У графа есть изолированные узлы: False
У графа есть петли: True
```

В отличие от `Cora` у набора `Facebook Page-Page` по умолчанию нет масок для обучения, валидации и тестирования. Мы можем произвольно создать маски с помощью функции `range()`:

```
# создаем маски
data.train_mask = range(18000)
data.val_mask = range(18001, 20000)
data.test_mask = range(20001, 22470)
```

Первый вывод подтверждает количество узлов и классов, которые мы видели в описании набора данных. Второй вывод сообщает нам, что в этом графе есть петли: некоторые страницы связаны сами с собой. Это удивительно, но на практике мы скоро увидим, что это не имеет особого значения.

Итак, у нас есть два графовых набора данных, которые мы будем использовать в следующем разделе, чтобы сравнить качество простой нейронной сети с качеством нашей первой графовой нейронной сети (GNN). Давайте реализуем их шаг за шагом.

Классификация узлов с помощью простых нейронных сетей

По сравнению с Клубом карате Закари, эти два набора данных включают новый тип информации – характеристики (признаки) узлов. Они содержат дополнительную информацию об узлах графа, например возраст, пол или интересы пользователя в социальной сети. В простой нейронной сети, также называемой **многослойным перцептроном** (multilayer perceptron), эти

эмбеддинги напрямую используются в модели для выполнения таких задач, как классификация узлов.

В этом разделе мы будем рассматривать информацию о характеристиках узлов как обычный табличный набор данных. На этом наборе данных мы обучим простую нейронную сеть, чтобы классифицировать узлы нашего графа. Обратите внимание, что данная архитектура не учитывает топологию сети. Мы постараемся исправить эту проблему в следующем разделе и сравним наши результаты.

Доступ к табличному набору, в котором записаны характеристики (признаки) узлов, можно легко получить с помощью созданного нами объекта `data`. Сначала мы преобразуем этот объект в обычный объект DataFrame библиотеки pandas, объединив `data.x` (содержащий признаки узла) и `data.y` (содержащий метку класса каждого узла, всего у нас будет семь классов). Начнем с набора данных `Cora`:

```
import pandas as pd

dataset = Planetoid(root=".", name="Cora")
data = dataset[0]

df_x = pd.DataFrame(data.x.numpy())
df_x['label'] = pd.DataFrame(data.y)
df_x
```

	0	1	2	3	4	5	6	7	8	9	...	1424	1425	1426	1427	1428	1429	1430	1431	1432	label
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
4	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2703	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3
2704	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3
2705	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3
2706	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3
2707	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3

2708 rows × 1434 columns

Если вы знакомы с машинным обучением, то, вероятно, уже знаете, как выглядит типичный набор данных, содержащий признаки и метки. Мы можем разработать простую модель многослойного перцептрона (multilayer perceptron – MLP) и обучить ее на массиве признаков `data.x` и массиве меток `data.y`.

Давайте напишем наш собственный класс MLP с четырьмя методами:

- `__init__()` для инициализации экземпляра класса;
- `.forward()` для выполнения прямого прохода;

- `.fit()` для обучения модели;
- `.test()` для оценки качества модели.

Прежде чем обучить нашу модель, мы должны определить основную метрику. Существует несколько метрик для задач многоклассовой классификации: **правильность** (accuracy), F1, **площадь под кривой рабочей характеристики приемника** (Area Under the Receiver Operating Characteristic Curve – AUC-ROC) и т. д. Для этого примера давайте реализуем простую правильность, которая определяется как доля правильных прогнозов. Это не лучшая метрика для многоклассовой классификации, но ее проще понять. Попробуйте потом заменить ее на свою метрику.

1. Сначала давайте зададим стартовое значение генератора псевдослучайных чисел. Затем импортируем класс `Linear` из модуля `torch.nn`, представляющий собой линейный слой. Он выполняет линейное преобразование входных данных. Потом импортируем модуль `torch.nn.functional`, используя алиас `F`. Этот модуль содержит различные функции активации, функции потерь и другие функции, которые могут использоваться при определении архитектуры нейронных сетей.

```
torch.manual_seed(0)
from torch.nn import Linear
import torch.nn.functional as F
```

2. Теперь напишем собственную функцию правильности.

```
def accuracy(y_pred, y_true):
    """Вычисляет правильность."""
    return torch.sum(y_pred == y_true) / len(y_true)
```

3. Мы создаем новый класс `MLP`, который наследует все методы и атрибуты класса `torch.nn.Module`:

```
class MLP(torch.nn.Module):
```

4. У метода `__init__()` три аргумента (`dim_in`, `dim_h` и `dim_out`), позволяющие задать количество нейронов во входном, скрытом и выходном слоях соответственно. Мы задаем два линейных слоя:

```
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        self.linear1 = Linear(dim_in, dim_h)
        self.linear2 = Linear(dim_h, dim_out)
```

5. Метод `.forward()` выполняет прямой проход. Входные данные проходят через первый линейный слой, затем применяется функция активации **Rectified Linear Unit** (**ReLU**), после чего результат проходит через второй линейный слой. Для классификации мы возвращаем логарифмический софтмакс (log softmax) итогового результата:

```
    def forward(self, x):
        x = self.linear1(x)
```

```
        x = torch.relu(x)
        x = self.linear2(x)
        return F.log_softmax(x, dim=1)
```

6. Метод `.fit()` отвечает за цикл обучения. Сначала мы инициализируем функцию потерь (кросс-энтропию) и оптимизатор Adam, которые будут использоваться в процессе обучения:

```
def fit(self, data, epochs):
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(self.parameters(),
                                 lr=0.01,
                                 weight_decay=5e-4)
```

7. Затем запускается обычный цикл обучения PyTorch. Помимо вычисления функции потерь, мы вычисляем правильность с помощью нашей функции `accuracy()`:

```
    self.train()
    for epoch in range(epochs+1):
        optimizer.zero_grad()
        out = self(data.x)
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        acc = accuracy(out[data.train_mask].argmax(dim=1),
                       data.y[data.train_mask])
        loss.backward()
        optimizer.step()
```

8. В том же цикле мы печатаем значения функции потерь и правильности для обучающей и валидационной выборок каждые 20 эпох:

```
        if(epoch % 20 == 0):
            val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
            val_acc = accuracy(out[data.val_mask].argmax(dim=1),
                               data.y[data.val_mask])
            print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: '
                  f'{loss:.3f} | Правильность на обуч. выборке: '
                  f'{acc*100:>5.2f}% \n| Функция потерь на валид. '
                  f'выборке: {val_loss:.2f} | Правильность на валид. '
                  f'выборке: {val_acc*100:.2f}%')
```

9. Метод `.test()` оценивает модель на тестовой выборке и возвращает оценку правильности. Обратите внимание, здесь мы дополнительно использовали декоратор `@torch.no_grad()` для отключения вычисления градиентов, чтобы уменьшить использование памяти:

```
@torch.no_grad()
def test(self, data):
    self.eval()
    out = self(data.x)
    acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
    return acc
```

Теперь, когда наш класс готов, мы можем создать экземпляр класса MLP, обучить модель на наборе Cora и оценить ее качество.

1. Создаем экземпляр класса MLP и печатаем информацию о нем, чтобы убедиться в том, что мы правильно задали слои:

```
# создаем модель MLP
mlp = MLP(dataset.num_features, 16, dataset.num_classes)
print(mlp)

MLP(
  (linear1): Linear(in_features=1433, out_features=16, bias=True)
  (linear2): Linear(in_features=16, out_features=7, bias=True)
)
```

2. Теперь обучаем модель, задав 100 эпох:

```
# обучаем
mlp.fit(data, epochs=100)
```

3. Ниже каждые 20 эпох печатаются значения функции потерь и метрики для обучающей и валидационной выборок:

```
MLP(
  (linear1): Linear(in_features=1433, out_features=16, bias=True)
  (linear2): Linear(in_features=16, out_features=7, bias=True)
)
Эпоха   0:
| Функция потерь на обуч. выборке: 1.959 | Правильность на обуч. выборке: 14.29%
| Функция потерь на валид. выборке: 2.00 | Правильность на валид. выборке: 12.40%
Эпоха  20:
| Функция потерь на обуч. выборке: 0.110 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 1.46 | Правильность на валид. выборке: 49.40%
Эпоха  40:
| Функция потерь на обуч. выборке: 0.014 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 1.44 | Правильность на валид. выборке: 51.00%
Эпоха  60:
| Функция потерь на обуч. выборке: 0.008 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 1.40 | Правильность на валид. выборке: 53.80%
Эпоха  80:
| Функция потерь на обуч. выборке: 0.008 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 1.37 | Правильность на валид. выборке: 55.40%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.009 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 1.34 | Правильность на валид. выборке: 54.60%
```

4. В итоге мы вычисляем оценку правильности для тестовых данных:

```
# тестируем
cora_mlp_acc = mlp.test(data)
print(f'\nПравильность MLP на тесте (Cora): {cora_mlp_acc*100:.2f}%')

Правильность MLP на тесте (Cora): 53.40%
```

Классификация узлов с помощью простых графовых нейронных сетей

Вместо непосредственного знакомства с хорошо известными архитектурами GNN давайте попробуем создать собственную модель, чтобы понять процесс обучения, лежащий в основе GNN. Однако сначала нам нужно вернуться к определению простого линейного слоя.

Базовый слой нейронной сети соответствует линейному преобразованию $h_A = x_A W^T$, где x_A – вектор входных данных узла A, а W – матрица весов. В PyTorch это уравнение можно реализовать с помощью функции `torch.mm()` или с помощью класса `nn.Linear`, который добавляет дополнительные параметры, например вектор смещений.

В наших графовых наборах векторы входных данных представляют собой характеристики (признаки) узлов. Это означает, что характеристики узла рассматриваются совершенно отдельно от характеристик соседних узлов. Этого недостаточно для хорошего понимания графа: в ходе анализа узла важен его контекст по аналогии с пикселем в картинке. Если вы посмотрите на группу пикселей, а не на один пиксель, вы сможете распознать края, узоры и т. д. Аналогично, чтобы исследовать узел, вам нужно посмотреть на его окрестность.

Пусть N_A – набор соседей узла A. Наш **линейный слой для графа** (graph linear layer) можно записать следующим образом:

$$h_A = \sum_{i \in N_A} x_i W^T.$$

Можно представить несколько вариантов этого уравнения. Например, у нас могла бы быть матрица весов W_1 для центрального узла и еще одна матрица весов W_2 – для его соседей. При этом обратите внимание, что у нас нет возможности создать отдельную матрицу весов для каждого соседнего узла. Это связано с тем, что в графах у каждого узла может быть разное количество соседей, и поэтому непрактично создавать отдельную матрицу для каждого из них.

В нейронных сетях нецелесообразно применять предыдущее уравнение к каждому узлу. Вместо этого мы используем матричные умножения, которые являются гораздо более эффективным способом обработки данных в нейронных сетях. Например, уравнение линейного слоя можно переписать как $H = XW^T$, где X – входная матрица (матрица признаков).

В нашем случае матрица смежности содержит информацию о связях (ребрах) между каждой парой узлов в графе. Умножение матрицы признаков на эту матрицу смежности непосредственно суммирует признаки соседних узлов для каждого узла. Это позволяет учесть контекст, формируемый соседями. Мы можем добавить петли (связи узла с самим собой) в матрицу смежности, чтобы в этой операции также учесть центральный узел. Давайте

назовем эту обновленную матрицу смежности $\tilde{A} = A + I$, где I – матрица единиц, представляющая петли. Наш графовый линейный слой можно переписать следующим образом:

$$H = \tilde{A}^T X W^T.$$

Давайте протестируем этот слой, реализовав его в PyTorch Geometric. Затем мы сможем использовать его как обычный слой при создании GNN.

1. Сначала мы создаем новый класс `VanillaGNNLayer`, который наследует все методы и аттрибуты класса `torch.nn.Module`:

```
class VanillaGNNLayer(torch.nn.Module):
```

2. У метода `__init__()` два аргумента `dim_in` и `dim_out`, размерность входного слоя и размерность выходного слоя соответственно. Мы добавляем базовое линейное преобразование без смещения.

```
    def __init__(self, dim_in, dim_out):
        super().__init__()
        self.linear = Linear(dim_in, dim_out, bias=False)
```

3. Выполняем две операции – линейное преобразование вектора признаков узлов `x` и затем матричное умножение обновленного вектора `x` на матрицу смежности:

```
    def forward(self, x, adjacency):
        x = self.linear(x)
        x = torch.sparse.mm(adjacency, x)
        return x
```

   Перед тем как создать простую GNN, нам нужно преобразовать индексы ребер из нашего набора данных (`data.edge_index`), записанные в формате координат, в плотную матрицу смежности. Кроме того, необходимо включить петли, в противном случае центральные узлы не будут учитываться в их собственных эмбеддингах.

4. Это легко реализовать с помощью функций `to_den_adj()` и `torch.eye()`. Функция `to_den_adj()` выполняет преобразование в плотную матрицу смежности. Функция `torch.eye()` создает единичную матрицу (матрицу, элементы которой по главной диагонали равны 1, а оставшиеся равны 0), которая затем используется для добавления петель в матрицу смежности.

```
from torch_geometric.utils import to_dense_adj

adjacency = to_dense_adj(data.edge_index)[0]
adjacency += torch.eye(len(adjacency))
adjacency

tensor([[1., 0., 0.,  ..., 0., 0., 0.],
        [0., 1., 1.,  ..., 0., 0., 0.],
```

```
[0., 1., 1.,  ..., 0., 0., 0.],
...,
[0., 0., 0.,  ..., 1., 0., 0.],
[0., 0., 0.,  ..., 0., 1., 1.],
[0., 0., 0.,  ..., 0., 1., 1.]])
```

Теперь, когда у нас есть выделенный слой и матрица смежности, реализация простой GNN будет очень похожа на реализацию MLP.

5. Мы создаем новый класс `VanillaGNN` с двумя простыми линейными слоями:

```
class VanillaGNN(torch.nn.Module):
    """Простая графовая нейронная сеть"""
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        self.gnn1 = VanillaGNNLayer(dim_in, dim_h)
        self.gnn2 = VanillaGNNLayer(dim_h, dim_out)
```

6. Далее мы выполняем те же самые операции, что и в классе `MLP`, используя наши новые слои, которые принимают в качестве дополнительных входных данных ранее вычисленную матрицу смежности:

```
    def forward(self, x, adjacency):
        h = self.gnn1(x, adjacency)
        h = torch.relu(h)
        h = self.gnn2(h, adjacency)
        return F.log_softmax(h, dim=1)

    def fit(self, data, epochs):
        criterion = torch.nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(self.parameters(),
                                     lr=0.01,
                                     weight_decay=5e-4)

        self.train()
        for epoch in range(epochs+1):
            optimizer.zero_grad()
            out = self(data.x, adjacency)
            loss = criterion(out[data.train_mask], data.y[data.train_mask])
            acc = accuracy(out[data.train_mask].argmax(dim=1),
                           data.y[data.train_mask])
            loss.backward()
            optimizer.step()

            if(epoch % 20 == 0):
                val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
                val_acc = accuracy(out[data.val_mask].argmax(dim=1),
                                   data.y[data.val_mask])
                print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: '
                      f'{loss:.3f} | Правильность на обуч. выборке: '
                      f'{acc*100:>5.2f}% \n| Функция потерь на валид. '
                      f'выборке: {val_loss:.2f} | Правильность на валид. '
                      f'выборке: {val_acc*100:.2f}%')
```

```
    @torch.no_grad()
    def test(self, data):
        self.eval()
        out = self(data.x, adjacency)
        acc = accuracy(out.argmax(dim=1)[data.test_mask], data.y[data.test_mask])
        return acc
```

7. Теперь наш класс готов, и мы можем создать экземпляр класса `VanillaGNN`, обучить модель на наборе `Cora` и оценить ее качество:

```
# создаем модель простой GNN
gnn = VanillaGNN(dataset.num_features, 16, dataset.num_classes)
print(gnn)

# обучаем
gnn.fit(data, epochs=100)

# тестируем
cora_gnn_acc = gnn.test(data)
print(f'\nПравильность GNN на тесте (Cora): {cora_gnn_acc*100:.2f}%')
```

```
VanillaGNN(
  (gnn1): VanillaGNNLayer(
    (linear): Linear(in_features=1433, out_features=16, bias=False)
  )
  (gnn2): VanillaGNNLayer(
    (linear): Linear(in_features=16, out_features=7, bias=False)
  )
)
Эпоха   0:
| Функция потерь на обуч. выборке: 1.991 | Правильность на обуч. выборке: 15.71%
| Функция потерь на валид. выборке: 2.11 | Правильность на валид. выборке: 9.40%
Эпоха  20:
| Функция потерь на обуч. выборке: 0.065 | Правильность на обуч. выборке: 99.29%
| Функция потерь на валид. выборке: 1.47 | Правильность на валид. выборке: 76.80%
Эпоха  40:
| Функция потерь на обуч. выборке: 0.014 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 2.11 | Правильность на валид. выборке: 75.40%
Эпоха  60:
| Функция потерь на обуч. выборке: 0.007 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 2.22 | Правильность на валид. выборке: 75.40%
Эпоха  80:
| Функция потерь на обуч. выборке: 0.004 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 2.20 | Правильность на валид. выборке: 76.80%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.003 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 2.19 | Правильность на валид. выборке: 77.00%

Правильность GNN на тесте (Cora): 76.60%
```

Теперь построим модели MLP и GNN для набора данных `Facebook Page-Page`.

```
# набор данных
dataset = FacebookPagePage(root=".")
data = dataset[0]
```

```
data.train_mask = range(18000)
data.val_mask = range(18001, 20000)
data.test_mask = range(20001, 22470)

# матрица смежности
adjacency = to_dense_adj(data.edge_index)[0]
adjacency += torch.eye(len(adjacency))
adjacency
```

```
tensor([[1., 0., 0.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        [0., 0., 1.,  ..., 0., 0., 0.],
        ...,
        [0., 0., 0.,  ..., 1., 0., 0.],
        [0., 0., 0.,  ..., 0., 1., 0.],
        [0., 0., 0.,  ..., 0., 0., 1.]])
```

```
# MLP
mlp = MLP(dataset.num_features, 16, dataset.num_classes)
print(mlp)
```

```
MLP(
  (linear1): Linear(in_features=128, out_features=16, bias=True)
  (linear2): Linear(in_features=16, out_features=4, bias=True)
)
```

```
mlp.fit(data, epochs=100)
```

```
Эпоха   0:
| Функция потерь на обуч. выборке: 1.401 | Правильность на обуч. выборке: 28.11%
| Функция потерь на валид. выборке: 1.40 | Правильность на валид. выборке: 28.91%
Эпоха  20:
| Функция потерь на обуч. выборке: 0.671 | Правильность на обуч. выборке: 73.47%
| Функция потерь на валид. выборке: 0.68 | Правильность на валид. выборке: 72.94%
Эпоха  40:
| Функция потерь на обуч. выборке: 0.579 | Правильность на обуч. выборке: 76.95%
| Функция потерь на валид. выборке: 0.61 | Правильность на валид. выборке: 74.89%
Эпоха  60:
| Функция потерь на обуч. выборке: 0.549 | Правильность на обуч. выборке: 78.20%
| Функция потерь на валид. выборке: 0.60 | Правильность на валид. выборке: 75.59%
Эпоха  80:
| Функция потерь на обуч. выборке: 0.533 | Правильность на обуч. выборке: 78.76%
| Функция потерь на валид. выборке: 0.60 | Правильность на валид. выборке: 75.39%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.520 | Правильность на обуч. выборке: 79.23%
| Функция потерь на валид. выборке: 0.60 | Правильность на валид. выборке: 75.39%
```

```
fb_mlp_acc = mlp.test(data)
print(f'\nПравильность MLP на тесте (Facebook): {fb_mlp_acc*100:.2f}%\n')
```

```
Правильность MLP на тесте (Facebook): 75.33%
```

```
# GCN
gnn = VanillaGNN(dataset.num_features, 16, dataset.num_classes)
print(gnn)
```

```
VanillaGNN(
  (gnn1): VanillaGNNLayer(
    (linear): Linear(in_features=128, out_features=16, bias=False)
  )
  (gnn2): VanillaGNNLayer(
    (linear): Linear(in_features=16, out_features=4, bias=False)
  )
)
```

```
gnn.fit(data, epochs=100)
```

```
Эпоха   0:
| Функция потерь на обуч. выборке: 176.683 | Правильность на обуч. выборке: 28.31%
| Функция потерь на валид. выборке: 173.10 | Правильность на валид. выборке: 28.41%
Эпоха  20:
| Функция потерь на обуч. выборке: 6.675 | Правильность на обуч. выборке: 79.69%
| Функция потерь на валид. выборке: 4.49 | Правильность на валид. выборке: 80.19%
Эпоха  40:
| Функция потерь на обуч. выборке: 2.284 | Правильность на обуч. выборке: 82.15%
| Функция потерь на валид. выборке: 1.60 | Правильность на валид. выборке: 83.64%
Эпоха  60:
| Функция потерь на обуч. выборке: 1.233 | Правильность на обуч. выборке: 83.91%
| Функция потерь на валид. выборке: 1.06 | Правильность на валид. выборке: 84.34%
Эпоха  80:
| Функция потерь на обуч. выборке: 2.959 | Правильность на обуч. выборке: 82.09%
| Функция потерь на валид. выборке: 1.89 | Правильность на валид. выборке: 82.04%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.926 | Правильность на обуч. выборке: 85.95%
| Функция потерь на валид. выборке: 0.82 | Правильность на валид. выборке: 85.69%
```

```
fb_gnn_acc = gnn.test(data)
print(f'\nПравильность GNN на тесте (Facebook): {fb_gnn_acc*100:.2f}%')
```

```
Правильность GNN на тесте (Facebook): 85.10%
```

Давайте сведем результаты наших экспериментов в одну таблицу.

```
data_dict = {'MLP': [f'{cora_mlp_acc*100:.2f}%',
                     f'{fb_mlp_acc*100:.2f}%'],
             'GNN': [f'{cora_gnn_acc*100:.2f}%',
                     f'{fb_gnn_acc*100:.2f}%']}
metrics_df = pd.DataFrame(data_dict,
                          index=['Cora', 'Facebook'])
metrics_df
```

	MLP	GNN
Cora	53.40%	76.60%
Facebook	75.33%	85.10%

Как мы видим, MLP демонстрирует низкую правильность на наборе данных `Cora`. Она работает получше на наборе данных `Facebook Page-Page`, но в обоих случаях ее все же превосходит наша простая GNN. Эти результаты

показывают важность включения топологической информации в качестве признаков узла. Вместо табличного набора данных наша GNN рассматривает всю окрестность каждого узла, что приводит к повышению правильности на 10–20 % в этих задачах. Эта архитектура все еще является довольно грубой, но она показывает нам, как ее можно усовершенствовать и построить модели с более высоким качеством прогнозирования.

Выводы

В этой главе мы узнали о недостающем звене между простыми нейронными сетями и GNN. Мы построили собственную архитектуру GNN, используя свою интуицию и немного линейной алгебры. Мы исследовали два популярных графовых набора данных из научной литературы, чтобы сравнить две наши архитектуры. Наконец, реализовали их в PyTorch и оценили их качество. Результат очевиден: даже наш простой вариант GNN полностью превосходит MLP на обоих наборах данных.

В главе 6 «Знакомство с графовыми сверточными нейронными сетями» мы усовершенствуем нашу стандартную архитектуру GNN, чтобы правильно нормализовать входные данные. Эта графовая сверточная нейронная сеть – невероятно эффективная базовая модель, которую мы будем использовать в оставшейся части книги. Мы оценим ее качество, используя те же наборы данных, и представим новую интересную задачу – регрессию узлов (node regression).

Дополнительное чтение

[1] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, “Collective Classification in Network Data”, AIMag, vol. 29, no. 3, p. 93, Sep. 2008. Доступ по ссылке https://ojs.aaai.org//index.php/aimagazine/article/view/2157.

[2] B. Rozemberczki, C. Allen, and R. Sarkar, Multi-Scale Attributed Node Embedding. arXiv, 2019. doi: 10.48550/ARXIV.1909.13021. Доступ по ссылке https://arxiv.org/abs/1909.13021.

Глава 6

Знакомство с графовыми сверточными нейронными сетями

Архитектура **графовой сверточной сети** (Graph Convolutional Network – GCN) лежит в основе GNN. Представленная Кипфом и Веллингом в 2017 году [1], она основана на идее создания эффективного варианта сверточных нейронных сетей (CNN), примененной к графам. Точнее, речь идет об аппроксимации операции свертки графа в ходе обработки сигналов на графе. Благодаря своей универсальности и простоте использования GCN стала самым популярным видом GNN в научной литературе. В более общем смысле это архитектура, которая позволяет получить надежную базовую модель для работы с графовыми данными.

В этой главе мы поговорим об ограничениях нашего простого GNN-слоя, предложенного ранее. Это поможет понять мотивацию использования GCN. Мы подробно расскажем, как работает GCN-слой и почему он работает лучше, чем наше решение. Мы проверим это утверждение, реализовав GCN и применив ее к наборам данных Cora и Facebook Page-Page с помощью PyTorch Geometric. Это должно в еще большей степени улучшить наши результаты.

Последний раздел посвящен новой задаче – **регрессии узлов** (node regression). Она не является широко распространенной в контексте использования GNN, однако весьма полезна, когда вы работаете с табличными данными. Если у вас есть возможность преобразовать набор табличных данных в граф, это позволит выполнить регрессию, помимо классификации.

К концу этой главы вы сможете реализовать GCN в PyTorch Geometric для задач классификации или регрессии. Благодаря линейной алгебре вы поймете, почему эта модель работает лучше, чем наша простая GNN. Наконец, вы узнаете, как визуализировать степени узлов и распределение плотности зависимой переменной.

В этой главе будут рассмотрены следующие темы:

- «Создание сверточного слоя графа»,
- «Сравнение сверточного и линейного слоев графа»,
- «Прогнозирование веб-трафика с помощью регрессии узлов».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter06.

Создание сверточного слоя графа

Сначала давайте поговорим о проблеме, которую мы не увидели в предыдущей главе. В отличие от табличных данных или данных изображений узлы не всегда имеют одинаковое количество соседей. Например, на рис. 6.1 узел 1 имеет 3 соседей, а узел 2 – только одного:

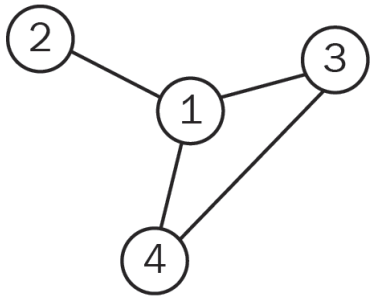


Рис. 6.1 ❖ Простой граф,
в котором узлы имеют разное количество соседей

Однако, если посмотреть на наш GNN-слой, мы не учитываем эту разницу в количестве соседей. Наш слой состоит из простой суммы без каких-либо коэффициентов нормализации. Вот как мы вычислили эмбеддинг узла, i:

$$h_i = \sum_{j \in \mathcal{N}_i} x_j W^T.$$

Представьте, что у узла A 1000 соседей, а у узла B – только один сосед: эмбеддинг h_A будет иметь гораздо большие значения, чем h_B. Это становится проблемой, потому что мы хотим сравнить эти эмбеддинги. Как мы можем выполнить содержательные сравнения эмбеддингов, если их значения так сильно отличаются друг от друга?

К счастью, есть простое решение: разделить эмбеддинг на количество соседей. Пусть deg(A) будет степенью узла A. Вот новая формула для GNN-слоя:

$$h_i = \frac{1}{\deg(i)} \sum_{j \in \mathcal{N}_i} x_j W^T.$$

Но как нам перевести это в операцию матричного умножения? Напомним, вот что мы получили для нашего простого GNN-слоя:

$$H = \tilde{A}^T X W^T.$$

Здесь $\tilde{A} = A + I$.

Единственное, чего не хватает в этой формуле, – это матрица, которая дала бы нам коэффициент нормализации. Ее можно получить благодаря матрице степеней, которая подсчитывает количество соседей для каждого узла. Ниже приведена матрица степеней для графа:

$$D = \begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}.$$

Вот та же матрица в NumPy:

```
import numpy as np

D = np.array([
    [3, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 2, 0],
    [0, 0, 0, 2]
])
```

По определению D дает вам степень каждого узла deg(i). Следовательно, обратная матрица D^{-1} напрямую дает нам коэффициенты нормализации $\frac{1}{\deg(i)}$:

$$D^{-1} = \begin{pmatrix} \frac{1}{3} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{pmatrix}.$$

Обратную матрицу можно напрямую вычислить с помощью функции `numpy.linalg.inv()`:

```
np.linalg.inv(D)
```

```
array([[0.33333333, 0.        , 0.        , 0.        ],
```

```
       [0.        , 1.        , 0.        , 0.        ],
       [0.        , 0.        , 0.5       , 0.        ],
       [0.        , 0.        , 0.        , 0.5       ]])
```

Это именно то, что мы искали. Вспомним, что для еще большей точности мы добавили в граф петли согласно формуле $\tilde{A} = A + I$. Аналогично мы должны добавить петли в матрицу степеней $\tilde{D} = D + I$. Итоговая матрица, которая нас действительно интересует, – это матрица $\tilde{D}^{-1} = (D + I)^{-1}$:

$$\tilde{D}^{-1} = \begin{pmatrix} \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} \end{pmatrix}.$$

В NumPy есть специальная функция `numpy.identity(n)`, позволяющая быстро создать единичную матрицу I с n измерениями. В этом примере у нас четыре измерения:

```
np.linalg.inv(D + np.identity(4))
```

```
array([[0.25      , 0.        , 0.        , 0.        ],
       [0.        , 0.5       , 0.        , 0.        ],
       [0.        , 0.        , 0.33333333, 0.        ],
       [0.        , 0.        , 0.        , 0.33333333]])
```

Теперь, когда у нас есть матрица коэффициентов нормализации, где нам разместить ее в формуле? Есть два варианта:

- $\tilde{D}^{-1}\tilde{A}XW^T$ выполняет нормализацию каждой строки матрицы признаков узлов;
- $\tilde{A}\tilde{D}^{-1}XW^T$ выполняет нормализацию каждого столбца матрицы признаков узлов.

Мы можем проверить их экспериментально, вычислив $\tilde{D}^{-1}\tilde{A}$ и $\tilde{A}\tilde{D}^{-1}$:

$$\tilde{D}^{-1}\tilde{A} = \begin{pmatrix} \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{3} & 0 & \frac{1}{3} & \frac{1}{3} \end{pmatrix};$$

$$\tilde{A}\tilde{D}^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{1}{4} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} \end{pmatrix} = \begin{pmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{4} & \frac{1}{2} & 0 & 0 \\ \frac{1}{4} & 0 & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{4} & 0 & \frac{1}{3} & \frac{1}{3} \end{pmatrix}.$$

Действительно в первом случае сумма каждой строки равна 1. Во втором случае сумма каждого столбца равна 1.

Умножение матриц можно выполнить с помощью функции `numpy.matmul()`. Кроме того, начиная с Python версии 3.5, вы можете использовать оператор

матричного умножения @. Давайте определим матрицу смежности A и воспользуемся этим оператором для умножения матриц:

```
A = np.array([
    [1, 1, 1, 1],
    [1, 1, 0, 0],
    [1, 0, 1, 1],
    [1, 0, 1, 1]
])

print(np.linalg.inv(D + np.identity(4)) @ A)
print()
print(A @ np.linalg.inv(D + np.identity(4)))
```

```
[[0.25       0.25       0.25       0.25      ]
 [0.5        0.5        0.         0.        ]
 [0.33333333 0.         0.33333333 0.33333333]
 [0.33333333 0.         0.33333333 0.33333333]]

[[0.25       0.5        0.33333333 0.33333333]
 [0.25       0.5        0.         0.        ]
 [0.25       0.         0.33333333 0.33333333]
 [0.25       0.         0.33333333 0.33333333]]
```

Мы получаем те же самые результаты, что и при умножении матриц вручную.

Итак, какой вариант нам следует использовать? Естественно, первый вариант выглядит более привлекательно, поскольку он хорошо нормализует признаки соседних узлов.

Однако Кипф и Веллинг [1] заметили, что признаки из узлов с большим количеством соседей распространяются (передаются) по графу очень легко (в отличие от признаков из более изолированных узлов). В оригинальной статье, посвященной GCN, авторы предложили гибридную нормализацию, чтобы уравновесить этот эффект. На практике они присваивают более высокие веса узлам с небольшим количеством соседей, используя следующую формулу:

$$H = \tilde{D}^{-\frac{1}{2}}\tilde{A}^{T}\tilde{D}^{-\frac{1}{2}}XW^{T}.$$

С точки зрения отдельных эмбеддингов эту операцию можно записать следующим образом:

$$h_i = \sum_{j \in \mathcal{N}_i} \frac{1}{\sqrt{\deg(i)}\sqrt{\deg(j)}} x_j W^{T}.$$

Это оригинальные формулы для реализации сверточного слоя графа. Как и в случае с нашим простым GNN-слоем, мы можем добавить эти слои, чтобы создать GCN. Давайте реализуем модель GCN и проверим, что она работает лучше, чем наши предыдущие модели.

Сравнение сверточных и линейных слоев графа

В предыдущей главе наша простая модель GNN превзошла модель Node2Vec, но как она работает в сравнении с GCN? В этом разделе мы сравним ее качество на наборах данных Cora и Facebook Page-Page.

По сравнению с простой GNN основная особенность GCN заключается в том, что она учитывает степени узла для взвешивания своих признаков. Перед реализацией собственной GCN давайте проанализируем степени узлов в обоих наборах данных. Эта информация является важной, поскольку она напрямую связана с работой GCN.

Из того, что мы знаем об этой архитектуре, мы ожидаем, что она будет работать лучше, когда степени узлов сильно различаются между собой. Если каждый узел имеет одинаковое количество соседей, эти архитектуры будут эквивалентны $\left(\sqrt{\deg(i)}\sqrt{\deg(i)} = \deg(i)\right)$.

1. Мы импортируем класс `Planetoid` из библиотеки PyTorch Geometric. Чтобы визуализировать степени узлов, мы также импортируем matplotlib и два дополнительных класса: `degree`, чтобы получить количество соседей каждого узла, и `Counter`, чтобы подсчитать количество узлов для каждой степени:

```
from torch_geometric.datasets import Planetoid
from torch_geometric.utils import degree
from collections import Counter
import matplotlib.pyplot as plt
```

2. Импортируем набор данных Cora и сохраняем в наборе `data`:

```
# импортируем набор данных из PyTorch Geometric
dataset = Planetoid(root=".", name="Cora")
data = dataset[0]
```

3. Мы создаем список степеней:

```
# получаем список степеней
degrees = degree(data.edge_index[0]).numpy()
```

4. Вычисляем количество узлов для каждой степени:

```
# вычисляем количество узлов для каждой степени
numbers = Counter(degrees)
```

5. Визуализируем результаты с помощью столбиковой диаграммы:

```
# столбиковая диаграмма
fig, ax = plt.subplots()
ax.set_xlabel('Степень узла')
```

```
    ax.set_ylabel('Количество узлов')
    plt.bar(numbers.keys(), numbers.values())
```

В итоге получаем следующий график.

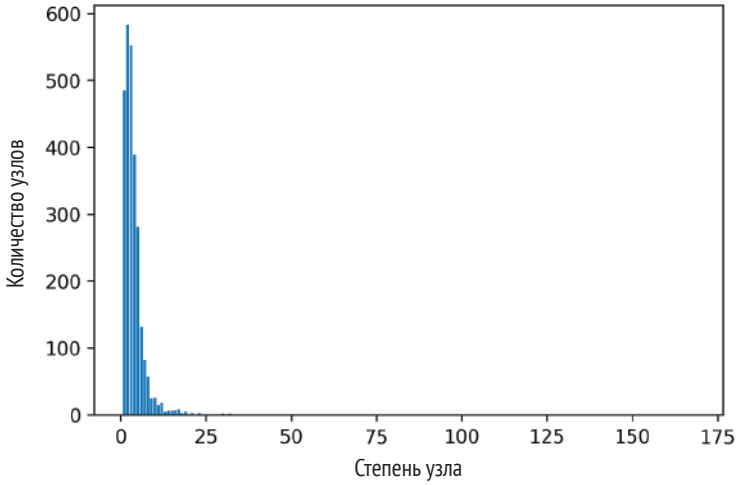


Рис. 6.2 ❖ Распределение степеней узлов в наборе данных Cora

Перед нами экспоненциальное распределение степеней узлов с длинным хвостом: степень узла варьирует от 1 соседа (485 узлов) до 168 соседей (1 узел)! В силу такого разброса степеней этот набор данных наглядно проиллюстрирует, как нормализация внутри GCN эффективно обработает этот дисбаланс.

Аналогичный график построим для набора данных `Facebook Page-Page`:

```
from torch_geometric.datasets import FacebookPagePage

# импортируем набор данных из PyTorch Geometric
dataset = FacebookPagePage(root=".")
data = dataset[0]

# создаем маски
data.train_mask = range(18000)
data.val_mask = range(18001, 20000)
data.test_mask = range(20001, 22470)

# создаем список степеней
degrees = degree(data.edge_index[0]).numpy()

# вычисляем количество узлов для каждой степени
numbers = Counter(degrees)

# столбиковая диаграмма
fig, ax = plt.subplots()
ax.set_xlabel('Степень узла')
```

```
ax.set_ylabel('Количество узлов')
plt.bar(numbers.keys(), numbers.values())
```

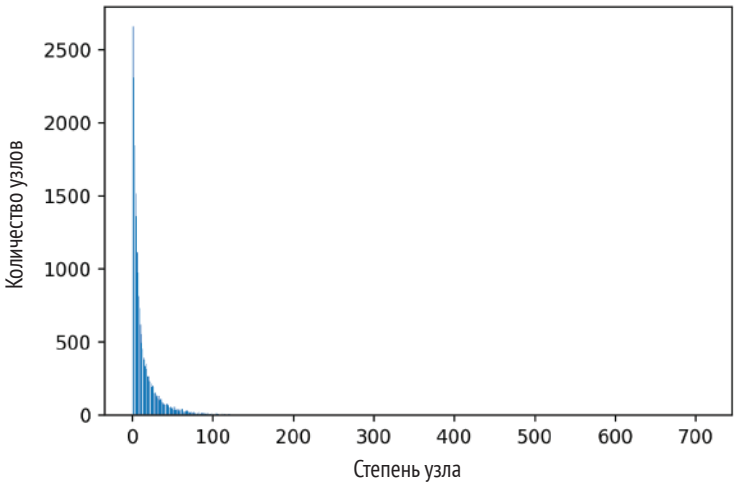


Рис. 6.3 ❖ Распределение степеней узлов в наборе данных Facebook Page-Page

Такое распределение степеней узлов выглядит еще более асимметричным: количество соседей варьирует от 1 до 709. Вновь по причине такого разброса степеней этот набор данных хорош для демонстрации возможностей GCN.

Мы могли бы создать собственный слой графа, однако в PyTorch Geometric можно воспользоваться уже готовым GCN-слоем. Давайте реализуем GCN и применим к набору данных `Cora`.

1. Мы импортируем PyTorch и GCN-слой из библиотеки PyTorch Geometric:

```
import torch
torch.manual_seed(1)
import torch.nn.functional as F
from torch_geometric.nn import GCNConv

dataset = Planetoid(root=".", name="Cora")
data = dataset[0]
```

2. Теперь напишем собственную функцию правильности.

```
def accuracy(y_pred, y_true):
    """Вычисляет правильность."""
    return torch.sum(y_pred == y_true) / len(y_true)
```

3. Мы создаем класс `GCN`, у метода `__init__()` три аргумента (`dim_in`, `dim_h` и `dim_out`), позволяющие задать размерности входного, скрытого и выходного слоев соответственно:

```
class GCN(torch.nn.Module):
    """Графовая сверточная нейросеть"""
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        self.gcn1 = GCNConv(dim_in, dim_h)
        self.gcn2 = GCNConv(dim_h, dim_out)
```

4. Метод `.forward()` идентичен методу `.forward()` класса `VanillaGNN`, только теперь мы используем два GCN-слоя. Для классификации возвращаем логарифмический софтмакс (log softmax) итогового результата:

```
    def forward(self, x, edge_index):
        h = self.gcn1(x, edge_index)
        h = torch.relu(h)
        h = self.gcn2(h, edge_index)
        return F.log_softmax(h, dim=1)
```

5. Метод `.fit()` идентичен методу `.fit()` класса `VanillaGNN` с теми же самыми параметрами оптимизатора `Adam` (темп обучения 0.01 и L2-регуляризация 0.0005):

```
    def fit(self, data, epochs):
        criterion = torch.nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(self.parameters(),
                                     lr=0.01,
                                     weight_decay=5e-4)

        self.train()
        for epoch in range(epochs+1):
            optimizer.zero_grad()
            out = self(data.x, data.edge_index)
            loss = criterion(out[data.train_mask], data.y[data.train_mask])
            acc = accuracy(out[data.train_mask].argmax(dim=1),
                           data.y[data.train_mask])
            loss.backward()
            optimizer.step()

            if(epoch % 20 == 0):
                val_loss = criterion(out[data.val_mask], data.y[data.val_mask])
                val_acc = accuracy(out[data.val_mask].argmax(dim=1),
                                   data.y[data.val_mask])
                print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: '
                      f'{loss:.3f} | Правильность на обуч. выборке: '
                      f'{acc*100:>5.2f}% \n| Функция потерь на валид. '
                      f'выборке: {val_loss:.2f} | Правильность на валид. '
                      f'выборке: {val_acc*100:.2f}%')
```

6. Метод `.test()` идентичен методу `.test()` ранее написанных классов:

```
    @torch.no_grad()
    def test(self, data):
```

```
        self.eval()
        out = self(data.x, data.edge_index)
        acc = accuracy(out.argmax(dim=1)[data.test_mask],
                       data.y[data.test_mask])
        return acc
```

7. Давайте создадим экземпляр класса и обучим модель GCN на наборе данных Cora, задав 100 эпох:

```
# создаем модель GCN
gcn = GCN(dataset.num_features, 16, dataset.num_classes)
print(gcn)
```

```
GCN(
  (gcn1): GCNConv(1433, 16)
  (gcn2): GCNConv(16, 7)
)
```

```
# обучаем
gcn.fit(data, epochs=100)
```

```
Эпоха   0:
| Функция потерь на обуч. выборке: 1.932 | Правильность на обуч. выборке: 15.71%
| Функция потерь на валид. выборке: 1.94 | Правильность на валид. выборке: 15.20%
Эпоха  20:
| Функция потерь на обуч. выборке: 0.099 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 0.75 | Правильность на валид. выборке: 77.80%
Эпоха  40:
| Функция потерь на обуч. выборке: 0.014 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 0.72 | Правильность на валид. выборке: 77.20%
Эпоха  60:
| Функция потерь на обуч. выборке: 0.015 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 0.71 | Правильность на валид. выборке: 77.80%
Эпоха  80:
| Функция потерь на обуч. выборке: 0.017 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 0.71 | Правильность на валид. выборке: 77.00%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.016 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 0.71 | Правильность на валид. выборке: 76.40%
```

8. Оцениваем качество модели на тестовой выборке:

```
# тестируем
cora_gcn_acc = gcn.test(data)
print(f'\nПравильность GCN на тесте (Cora): {cora_gcn_acc*100:.2f}%\n')
```

```
Правильность GCN на тесте (Cora): 79.70%
```

Теперь обучим модель GCN на наборе данных Facebook Page-Page и также оценим ее качество.

```
# импортируем набор данных из PyTorch Geometric
dataset = FacebookPagePage(root=".")
data = dataset[0]
data.train_mask = range(18000)
data.val_mask = range(18001, 20000)
data.test_mask = range(20001, 22470)

# создаем модель GCN
gcn = GCN(dataset.num_features, 16, dataset.num_classes)
print(gcn)
```

```
GCN(
  (gcn1): GCNConv(128, 16)
  (gcn2): GCNConv(16, 4)
)
```

```
# обучаем
gcn.fit(data, epochs=100)
```

```
Эпоха   0:
| Функция потерь на обуч. выборке: 1.463 | Правильность на обуч. выборке: 20.72%
| Функция потерь на валид. выборке: 1.45 | Правильность на валид. выборке: 20.71%
Эпоха  20:
| Функция потерь на обуч. выборке: 0.443 | Правильность на обуч. выборке: 84.64%
| Функция потерь на валид. выборке: 0.43 | Правильность на валид. выборке: 85.29%
Эпоха  40:
| Функция потерь на обуч. выборке: 0.323 | Правильность на обуч. выборке: 89.59%
| Функция потерь на валид. выборке: 0.31 | Правильность на валид. выборке: 90.20%
Эпоха  60:
| Функция потерь на обуч. выборке: 0.278 | Правильность на обуч. выборке: 91.36%
| Функция потерь на валид. выборке: 0.27 | Правильность на валид. выборке: 91.55%
Эпоха  80:
| Функция потерь на обуч. выборке: 0.254 | Правильность на обуч. выборке: 92.28%
| Функция потерь на валид. выборке: 0.26 | Правильность на валид. выборке: 92.75%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.238 | Правильность на обуч. выборке: 92.82%
| Функция потерь на валид. выборке: 0.25 | Правильность на валид. выборке: 92.75%
```

```
# тестируем
fb_gcn_acc = gcn.test(data)
print(f'\nПравильность GCN на тесте (Facebook): {fb_gcn_acc*100:.2f}%\n')
```

```
Правильность GCN на тесте (Facebook): 91.70%
```

Мы можем объяснить более высокие оценки правильности большим разбросом степеней узлов в обоих наборах данных. За счет нормализации признаков и учета количества соседей узлов GCN приобретает большую гибкость и может хорошо работать с различными типами графов.

Однако классификация узлов не единственная задача, которую могут выполнять GNN. В следующем разделе мы рассмотрим новый тип приложений, который редко освещается в литературе.

Прогнозирование веб-трафика с помощью регрессии узлов

В машинном обучении **регрессия** относится к прогнозированию непрерывных значений. Ее часто противопоставляют **классификации**, цель которой состоит в том, чтобы найти корректные классы (которые являются дискретными, а не непрерывными значениями). Применительно к графовым данным их аналогами являются классификация узлов и регрессия узлов. В этом разделе мы попытаемся предсказать для каждого узла непрерывное значение вместо класса.

Набор данных, который мы будем использовать, – это сеть Wikipedia (GNU General Public License v3.0), представленная Rozemberckzi et al. в 2019 году [2]. Она состоит из трех page-page-сетей: `chameleons` (2277 узлов и 31 421 ребро), `crocodiles` (11 631 узел и 170 918 ребер) и `squirrels` (5201 узел и 198 493 ребра). В этих наборах данных узлы представляют собой статьи, а ребра – взаимные связи между ними. Признаки узла фиксируют наличие тех или иных слов в статьях. Задача состоит в том, чтобы спрогнозировать среднемесячный трафик в декабре 2018 года.

В этом разделе мы применим GCN для прогнозирования этого трафика применительно к набору данных `chameleons`.

1. Мы импортируем класс `WikipediaNetwork` и с его помощью загружаем набор данных chameleons. Далее применяем функцию преобразования `RandomNodeSplit()` для случайного создания масок (по ним будут созданы обучающее и тестовое подмножества):

```
from torch_geometric.datasets import WikipediaNetwork
import torch_geometric.transforms as T

dataset = WikipediaNetwork(
    root=".", name="chameleon",
    transform=T.RandomNodeSplit(num_val=200, num_test=500)
)
data = dataset[0]
```

2. Далее печатаем информацию о наборе данных:

```
# печатаем информацию о наборе данных
print(f'Набор данных: {dataset}')
print('---------------')
print(f'Количество графов: {len(dataset)}')
print(f'Количество узлов: {data.x.shape[0]}')
print(f'Количество признаков: {dataset.num_features}')
print(f'Количество классов: {dataset.num_classes}')

# печатаем информацию о графе
print(f'\nГраф:')
print('------')
```

```
print(f'Ребра являются ориентированными: {data.is_directed()}')
print(f'У графа есть изолированные узлы: {data.has_isolated_nodes()}')
print(f'У графа есть петли: {data.has_self_loops()}')
```

```
Набор данных: WikipediaNetwork()
---------------
Количество графов: 1
Количество узлов: 2277
Количество признаков: 2325
Количество классов: 5

Граф:
------
Ребра являются ориентированными: True
У графа есть изолированные узлы: False
У графа есть петли: True
```

3. С нашим набором данных возникла проблема: в выводе указано, что у нас пять классов. Однако мы хотим выполнить регрессию узлов, а не классификацию. Итак, что случилось?

   Фактически эти пять классов представляют собой бины непрерывной переменной, которые мы хотим предсказать. К сожалению, эти метки не те, которые нам нужны: придется поменять их вручную. Сначала давайте загрузим файл *wikipedia.zip* со следующей страницы: https://snap.stanford.edu/data/wikipedia-article-networks.html. После разархивирования файла импортируем pandas и используем его для получения значений зависимой переменной:

```
from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile

url = 'https://snap.stanford.edu/data/wikipedia.zip'
with urlopen(url) as zurl:
    with ZipFile(BytesIO(zurl.read())) as zfile:
        zfile.extractall('.')

import pandas as pd

df = pd.read_csv('wikipedia/chameleon/musae_chameleon_target.csv')
```

4. Мы применяем логарифмирование зависимой переменной, используя функцию `np.log10()`, поскольку задача состоит в том, чтобы прогнозировать прологарифмированный среднемесячный трафик:

```
values = np.log10(df['target'])
```

5. Переопределяем `data.y` как тензор непрерывных значений, полученных на предыдущем шаге. Обратите внимание, что в этом примере эти значения не нормализованы, хотя это обычно является хорошей практикой. Мы не будем приводить это здесь для простоты изложения:

```
data.y = torch.tensor(values)
data.y

tensor([2.2330, 3.9079, 3.9329,  ..., 1.9956, 4.3598, 2.4409],
       dtype=torch.float64)
```

Давайте снова визуализируем распределение степеней узлов, как мы делали это раньше.

```
# получаем список степеней
degrees = degree(data.edge_index[0]).numpy()

# вычисляем количество узлов для каждой степени
numbers = Counter(degrees)

# столбиковая диаграмма
fig, ax = plt.subplots()
ax.set_xlabel('Степень узла')
ax.set_ylabel('Количество узлов')
plt.bar(numbers.keys(), numbers.values())
```

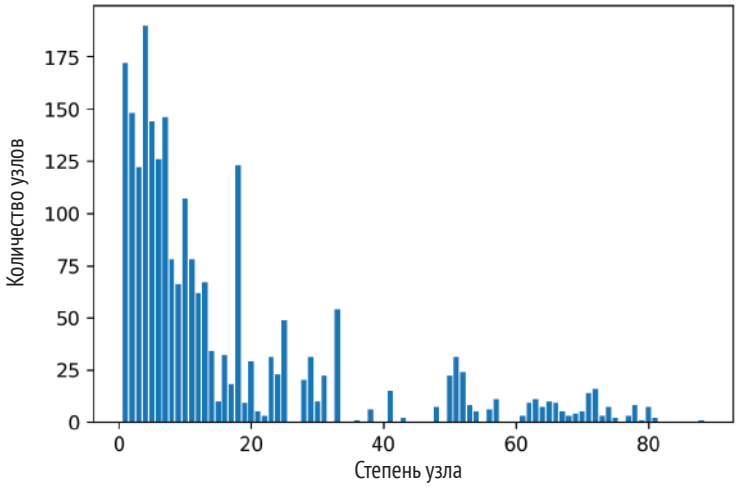


Рис. 6.4 ❖ Распределение степеней узлов в наборе данных Wikipedia Network

У этого распределения, в отличие от предыдущих, менее длинный правый хвост, но оно сохраняет аналогичную форму: у большинства узлов есть один или несколько соседей, однако некоторые из них действуют как «концентраторы» и могут соединять более 80 узлов.

В случае регрессии узлов распределение степеней узлов не единственный тип распределения, который мы должны проверить, распределение нашей зависимой переменной также важно. Действительно ненормальное распределение (например, распределение степеней узла) предсказать труднее. Мы можем воспользоваться библиотекой Seaborn для визуализации распределе-

ния зависимой переменной и сравнения его с нормальным распределением, представленным объектом `scipy.stats.norm`:

```
import seaborn as sns
from scipy.stats import norm

df['target'] = values
fig = sns.distplot(df['target'], fit=norm)
```

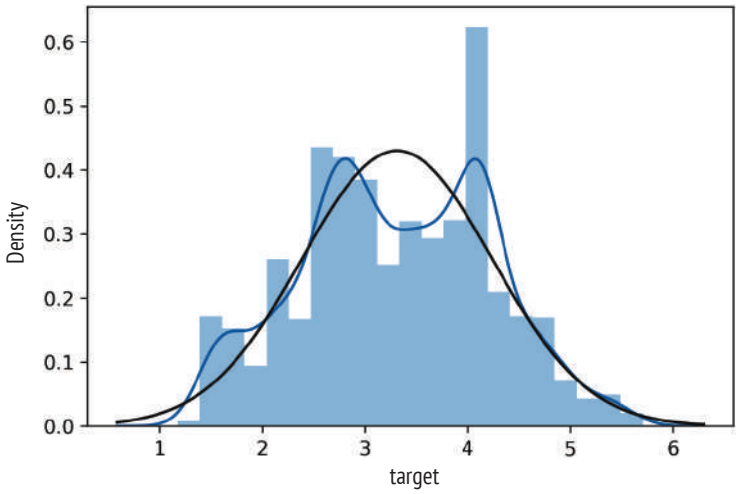


Рис. 6.5 ❖ Распределение зависимой переменной в наборе данных Wikipedia Network

Это распределение не является в строгом смысле нормальным, но оно не похоже и на экспоненциальное, как предыдущие распределения степеней узла. Можно ожидать, что наша модель хорошо спрогнозирует эти значения.

Давайте пошагово реализуем модель с помощью PyTorch Geometric.

1. Мы пишем класс `GCN` и начинаем с метода `__init__()`. На этот раз у нас будет три `GCNConv`-слоя с уменьшающимся количеством нейронов. Идея, стоящая за этой архитектурой кодировщика, состоит в том, чтобы заставить модель выбирать наиболее релевантные признаки для прогнозирования значений зависимой переменной. Кроме того, мы добавили линейный слой для вывода прогноза, который не ограничен числами в диапазоне от 0 до –1 или 1:

```
class GCN(torch.nn.Module):
    """Графовая сверточная сеть"""
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        self.gcn1 = GCNConv(dim_in, dim_h*4)
        self.gcn2 = GCNConv(dim_h*4, dim_h*2)
        self.gcn3 = GCNConv(dim_h*2, dim_h)
        self.linear = torch.nn.Linear(dim_h, dim_out)
```

2. Метод `.forward()` включает новые `GCNConv`-слои и слои `nn.Linear`. Функция логарифмического софтмакса теперь нам не нужна, поскольку мы не предсказываем класс:

```
def forward(self, x, edge_index):
    h = self.gcn1(x, edge_index)
    h = torch.relu(h)
    h = F.dropout(h, p=0.5, training=self.training)
    h = self.gcn2(h, edge_index)
    h = torch.relu(h)
    h = F.dropout(h, p=0.5, training=self.training)
    h = self.gcn3(h, edge_index)
    h = torch.relu(h)
    h = self.linear(h)
    return h
```

3. Главным нововведением в методе `.fit()` является функция `F.mse_loss()`, которая заменяет кросс-энтропию, используемую в задачах классификации. **Среднеквадратичная ошибка** (Mean Squared Error – MSE) будет нашей главной метрикой. Она соответствует усредненной разности между фактическим и спрогнозированным значениями зависимой переменной:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2.$$

Программно это реализовано следующим образом:

```
def fit(self, data, epochs):
    optimizer = torch.optim.Adam(self.parameters(),
                                 lr=0.02,
                                 weight_decay=5e-4)

    self.train()
    for epoch in range(epochs+1):
        optimizer.zero_grad()
        out = self(data.x, data.edge_index)
        loss = F.mse_loss(out.squeeze()[data.train_mask],
                          data.y[data.train_mask].float())
        loss.backward()
        optimizer.step()
        if epoch % 20 == 0:
            val_loss = F.mse_loss(out.squeeze()[data.val_mask],
                                  data.y[data.val_mask])

            print(f"Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: "
                  f"{loss:.5f} | Функция потерь на валид. выборке: "
                  f"{val_loss:.5f}")
```

4. MSE также включена в метод `.test()`:

```
def test(self, data):
        self.eval()
```

```
        out = self(data.x, data.edge_index)
        return F.mse_loss(out.squeeze()[data.test_mask],
                          data.y[data.test_mask].float())
```

5. Мы создаем экземпляр класса GCN со 128 нейронами в скрытом слое и 1 нейроном в выходном слое (значением зависимой переменной). Обучаем модель, задав 200 эпох:

```
# создаем модель GCN
gcn = GCN(dataset.num_features, 128, 1)
print(gcn)
```

```
GCN(
  (gcn1): GCNConv(2325, 512)
  (gcn2): GCNConv(512, 256)
  (gcn3): GCNConv(256, 128)
  (linear): Linear(in_features=128, out_features=1, bias=True)
)
```

```
# обучаем
gcn.fit(data, epochs=200)
```

```
Эпоха   0:
| Функция потерь на обуч. выборке: 12.81936 | Функция потерь на валид. выборке: 13.00659
Эпоха  20:
| Функция потерь на обуч. выборке: 11.71822 | Функция потерь на валид. выборке: 11.76676
Эпоха  40:
| Функция потерь на обуч. выборке: 10.29417 | Функция потерь на валид. выборке: 10.34435
Эпоха  60:
| Функция потерь на обуч. выборке: 8.67988 | Функция потерь на валид. выборке: 8.72097
Эпоха  80:
| Функция потерь на обуч. выборке: 4.03177 | Функция потерь на валид. выборке: 4.00314
Эпоха 100:
| Функция потерь на обуч. выборке: 1.94211 | Функция потерь на валид. выборке: 2.00301
Эпоха 120:
| Функция потерь на обуч. выборке: 0.97431 | Функция потерь на валид. выборке: 1.10884
Эпоха 140:
| Функция потерь на обуч. выборке: 0.73444 | Функция потерь на валид. выборке: 0.89813
Эпоха 160:
| Функция потерь на обуч. выборке: 0.59117 | Функция потерь на валид. выборке: 0.80247
Эпоха 180:
| Функция потерь на обуч. выборке: 0.52785 | Функция потерь на валид. выборке: 0.80964
Эпоха 200:
| Функция потерь на обуч. выборке: 0.51481 | Функция потерь на валид. выборке: 0.64949
```

```
# тестируем
loss = gcn.test(data)
print(f'\nФункция потерь модели GCN на тесте: {loss:.5f}\n')
```

```
Функция потерь модели GCN на тесте 0.72098
```

Функция потерь MSE сама по себе не является наиболее легкоинтерпретируемой метрикой. Мы можем получить более содержательные результаты, используя две следующие метрики:

- **RMSE**, которая измеряет среднюю величину ошибки:

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2};$$

- **средняя абсолютная ошибка** (Mean Absolute Error – MAE) соответствует усредненной абсолютной разности между фактическим и спрогнозированным значениями зависимой переменной:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|.$$

Теперь реализуем их на Python.

1. Мы можем напрямую импортировать MSE и MAE из библиотеки scikit-learn:

```
from sklearn.metrics import (mean_squared_error,
                             mean_absolute_error)
```

2. Преобразуем тензоры PyTorch для хранения прогнозов в массивы NumPy с помощью `.detach().numpy()`:

```
out = gcn(data.x, data.edge_index)
y_pred = out.squeeze()[data.test_mask].detach().numpy()
mse = mean_squared_error(data.y[data.test_mask], y_pred)
mae = mean_absolute_error(data.y[data.test_mask], y_pred)
```

3. Теперь вычисляем MSE и MAE с помощью импортированных функций. RMSE вычисляем как квадратный корень MSE, используя функцию `np.sqrt()`:

```
print('=' * 43)
print(f'MSE = {mse:.4f} | RMSE = {np.sqrt(mse):.4f} | MAE = {mae:.4f}')
print('=' * 43)
```

```
===========================================
MSE = 0.7210 | RMSE = 0.8491 | MAE = 0.6828
===========================================
```

Эти метрики полезны для сравнения различных моделей, но интерпретировать MSE и RMSE может быть сложно.

Лучшим инструментом для визуализации результатов нашей модели является диаграмма рассеяния, на которой по горизонтальной оси отложены фактические значения, а по вертикальной оси – спрогнозированные значения. У Seaborn есть специальная функция `regplot()` для этого типа визуализации:

```
fig = sns.regplot(x=data.y[data.test_mask].numpy(), y=y_pred)
fig.set(xlabel='Фактические значения',
        ylabel='Спрогнозированные значения')
```

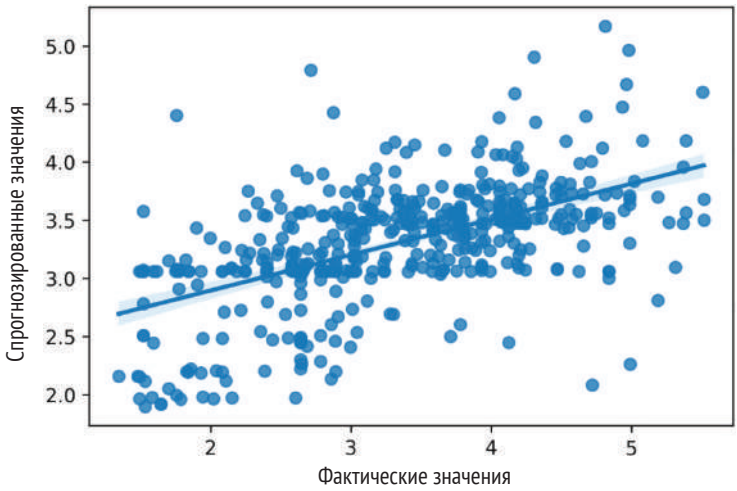


Рис. 6.6 ❖ Диаграмма рассеяния, по оси x – фактические значения, по оси y – спрогнозированные значения

В этом примере у нас нет базовой модели, с которой можно было бы сравнить использованную нами модель GCN, но это достойный прогноз с небольшим количеством выбросов. Это приемлемо для большого числа задач, несмотря на минималистичный набор данных. Если необходимо улучшить эти результаты, можно настроить гиперпараметры и провести дополнительный анализ ошибок, чтобы понять, откуда берутся выбросы.

Выводы

В этой главе мы улучшили наш простой GNN-слой, чтобы корректно нормализовать признаки. Мы узнали про GCN-слой и интеллектуальную нормализацию. Сравнили эту новую архитектуру с Node2Vec и нашей простой моделью GNN на наборах данных `Cora` и `Facebook Page-Page`. Благодаря этому процессу нормализации GCN в обоих случаях получила самые высокие оценки правильности с большим отрывом от остальных моделей. Наконец, мы применили модель GCN для решения задачи регрессии узлов (на примере Wikipedia Network) и научились решать эту новую задачу.

В главе 7 «Графовые нейронные сети с механизмом внимания» мы пойдем еще дальше и научимся выделять соседние узлы на основе их важности. Мы узнаем, как автоматически взвешивать характеристики узла с помощью процесса, называемого самовниманием. Как мы увидим далее, это улучшит качество прогнозов по сравнению с архитектурой GCN.

Дополнительное чтение

[1] T. N. Kipf and M. Welling, Semi-Supervised Classification with Graph Convolutional Networks. arXiv, 2016. DOI: 10.48550/ARXIV.1609.02907. Доступ по ссылке https://arxiv.org/abs/1609.02907.

[2] B. Rozemberczki, C. Allen, and R. Sarkar, Multi-scale Attributed Node Embedding. arXiv, 2019. DOI: 10.48550/ARXIV.1909.13021. Доступ по ссылке https://arxiv.org/abs/1909.13021.

Глава 7

Графовые нейронные сети с механизмом внимания

Графовая нейронная сеть с механизмом самовнимания (Graph Attention Network – GAT) является теоретическим улучшением по сравнению с GCN. Вместо статических коэффициентов нормализации они предлагают весовые коэффициенты, рассчитываемые с помощью процесса, называемого **самовниманием** (self-attention). Тот же процесс лежит в основе одной из самых успешных архитектур глубокого обучения: **трансформера** (transformer), который приобрел популярность благодаря **BERT** и **GPT-3**. Графовые нейронные сети с механизмом самовнимания, представленные Величковичем совместно с коллегами в 2017 году (Velickovic et al., 2017), стали одной из самых популярных архитектур GNN благодаря превосходному качеству «из-под коробки».

В этой главе мы поэтапно выясним, как работает слой внимания графа. На самом деле слой внимания графа – это идеальный пример для понимания того, как самовнимание работает в целом. Эта теоретическая основа позволит нам реализовать слой внимания графа с нуля в NumPy. Мы самостоятельно получим матрицы, чтобы понять, как вычисляются их значения на каждом шаге.

В последнем разделе мы применим GAT к двух наборам данных для задачи классификации узлов: набору `Cora` и новому набору `CiteSeer`. Как и предполагалось в предыдущей главе, это будет хорошая возможность проанализировать наши результаты немного глубже. Наконец, мы сравним качество прогнозов GAT с качеством прогнозов GCN.

К концу этой главы вы сможете с нуля реализовать слой внимания графа и GAT в **PyTorch Geometric** (**PyG**). Вы узнаете о различиях между этой архитектурой и GCN. Кроме того, вы освоите инструмент анализа ошибок графовых данных.

В этой главе будут рассмотрены следующие темы:

- «Знакомство со слоем внимания графа»,
- «Реализация слоя внимания графа в NumPy»,
- «Реализация GAT в PyTorch Geometric».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter07.

Знакомство со слоем внимания графа

Основная идея GAT заключается в том, что некоторые узлы более важны, чем другие. Фактически она уже была основной идеей сверточного слоя графа: узлы с небольшим количеством соседей были более важными, чем другие, благодаря коэффициенту нормализации $\frac{1}{\sqrt{\deg(i)}\sqrt{\deg(j)}}$. Этот подход имеет ограничения, поскольку он учитывает только степени узлов. С другой стороны, цель слоя внимания графа состоит в том, чтобы создавать весовые коэффициенты, которые также учитывают важность признаков узла.

Давайте назовем наши весовые коэффициенты **оценками внимания** (attention scores), пусть α_{ij} будет оценкой внимания, определяющей важность связи между узлами i и j. Мы можем определить оператор внимания графа следующим образом:

$$h_i = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} x_j.$$

Важная особенность GAT заключается в том, что оценки внимания вычисляются неявно путем сравнения входных данных друг с другом (отсюда и название *само*внимание). В этом разделе мы выясним, как вычислить эти оценки внимания в четыре этапа, а также как улучшить слой внимания графа, таким образом, будут разобраны следующие пункты:

- линейное преобразование;
- функция активации;
- нормализация с помощью softmax;
- многоголовое внимание;
- улучшенный слой внимания графа.

Прежде всего давайте посмотрим, чем линейное преобразование отличается от предыдущих архитектур.

Линейное преобразование

Оценка внимания представляет важность связи между центральным узлом i и соседом j. Как указывалось ранее, для вычисления оценки внимания требуются признаки обоих узлов. В слое внимания графа она представляет собой конкатенацию скрытых векторов $\mathbf{W}x_i$ и $\mathbf{W}x_j$, $[\mathbf{W}x_i || \mathbf{W}x_j]$. Здесь $\mathbf{W}$ – это классическая матрица общих весов для вычисления скрытых векторов. К этому результату применяется дополнительное линейное преобразование с помощью специальной обучаемой матрицы весов W_{att}. Во время обучения эта матрица «выучивает» веса для получения коэффициентов внимания a_{ij}. Этот процесс можно подытожить с помощью следующей формулы:

$$a_{ij} = W_{att}^{T}[\mathbf{W}x_i || \mathbf{W}x_j].$$

Как и в традиционных нейронных сетях, этот результат передается функции активации.

Функция активации

Нелинейность является важным компонентом нейронных сетей для аппроксимации нелинейных целевых функций. Такие функции нельзя передать простым комбинированием линейных слоев, поскольку конечный результат такой комбинации все равно будет вести себя как один линейный слой.

В официальной реализации (https://github.com/PetarV-/GAT/blob/master/utils/layers.py) авторы выбрали функцию активации Leaky Rectified Linear Unit (ReLU) (см. рис. 7.1). Эта функция устраняет проблему умирающего ReLU (*dying ReLU problem*), когда нейроны ReLU выдают только нулевые значения:

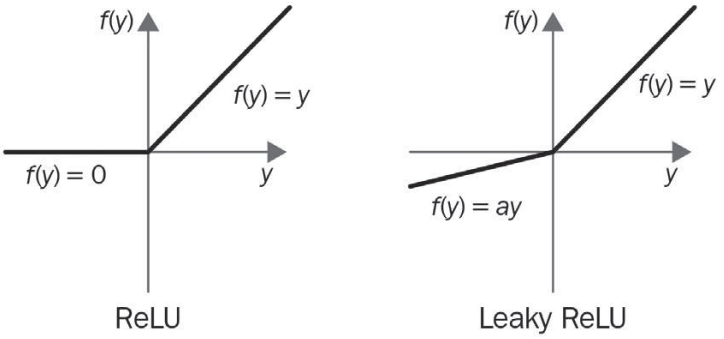


Рис. 7.1 ❖ Сравнение функций ReLU и LeakyReLU

Применяем функцию Leaky ReLU к результату предыдущего шага:

$$e_{ij} = \text{LeakyReLU}(a_{ij}).$$

Однако здесь мы сталкиваемся с новой проблемой: полученные значения не нормализованы!

Нормализация с помощью softmax

Нам необходимо сравнивать разные оценки внимания, а это значит, что нам нужны нормализованные значения, лежащие в одном и том же масштабе. В машинном обучении для этой цели обычно используется функция softmax. Давайте назовем $\mathcal{N}_i$ соседями узла i, включая сам узел i:

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}.$$

Результат этой операции дает нам итоговые оценки внимания α_{ij}. Но есть еще одна проблема: самовнимание не очень стабильно.

Многоголовое внимание

Эту проблему заметили Васвани с коллегами в оригинальной статье, посвященной трансформерам (Vaswani et al., 2017). Предложенное ими решение состоит в вычислении нескольких эмбеддингов с собственными оценками внимания вместо одного. Этот метод называется **многоголовым вниманием** или **множественным вниманием** (multi-head attention).

Реализация проста, так как нам нужно просто повторить три предыдущих шага несколько раз. Каждый экземпляр создает эмбеддинг h_i^k, где k – индекс головы внимания. Существует два способа объединения полученных результатов:

- **усреднение** (averaging): здесь мы суммируем разные эмбеддинги и нормализуем результат, поделив его на количество голов внимания:

$$h_i = \frac{1}{n}\sum_{k=1}^{n} h_i^k = \frac{1}{n}\sum_{k=1}^{n}\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k x_j;$$

- **конкатенация** (concatenation): здесь мы объединяем разные эмбеддинги, в результате чего получаем матрицу большего размера:

$$h_i = \Big\|_{k=1}^{n} h_i^k = \Big\|_{k=1}^{n} \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k x_j.$$

На практике существует простое правило, позволяющее понять, какой способ из них использовать: мы выбираем конкатенацию, когда работаем со скрытым слоем, и усреднение, когда работаем с последним слоем сети. Весь процесс можно подытожить следующей схемой:

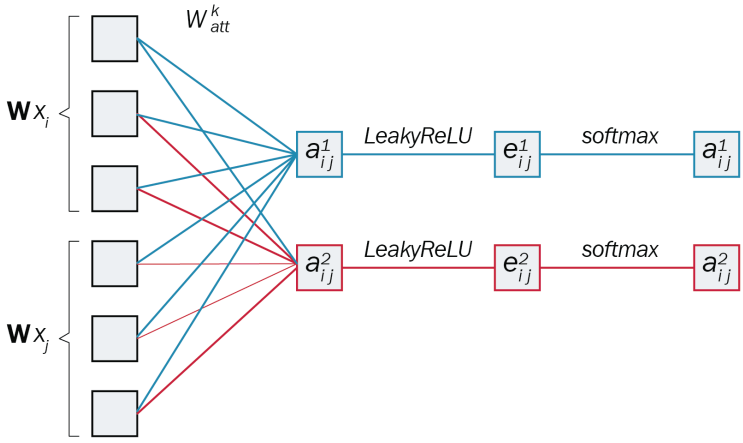


Рис. 7.2 ❖ Вычисление оценок внимания с помощью многоголового (множественного) внимания

Это все, что нужно знать о теоретическом аспекте слоя внимания графа. Однако с момента его создания в 2017 году было предложено улучшение.

Улучшенный слой внимания графа

В своей работе 2021 года Шакед Броуди совместно с коллегами (Brody et al., 2021) утверждал, что слой внимания графа поддерживает только статический тип внимания. И это проблема, поскольку существуют простые графовые задачи, которые невозможно выразить с помощью GAT. Поэтому они представили улучшенную версию под названием GATv2, которая предлагает гораздо более выразительное динамическое внимание.

Их решение состоит в изменении порядка операций. Матрица весов W применяется после конкатенации, а матрица весов внимания W_{att} – после функции *LeakyReLU*. Ниже приведен исходный **оператор внимания графа** (Graph Attentional Operator), тоже GAT:

$$\alpha_{ij} = \frac{\exp(LeakyReLU(W_{att}^T[\mathbf{W}x_i||\mathbf{W}x_j]))}{\sum_{k\in\mathcal{N}_i}\exp(LeakyReLU(W_{att}^T[\mathbf{W}x_i||\mathbf{W}x_k]))}.$$

А это уже модифицированный оператор GATv2, предложенный Броуди совместно с коллегами:

$$\alpha_{ij} = \frac{\exp(W_{att}^T LeakyReLU(\mathbf{W}[x_i||x_j]))}{\sum_{k\in\mathcal{N}_i}\exp(W_{att}^T LeakyReLU(\mathbf{W}[x_i||x_k]))}.$$

Какой из операторов нам следует использовать? По мнению Броуди и его коллег, GATv2 постоянно превосходит GAT, и поэтому ему следует отдать

предпочтение. Кроме теоретического доказательства, они также провели несколько экспериментов, чтобы продемонстрировать качество GATv2 по сравнению с исходным GAT. В оставшейся части этой главы мы рассмотрим оба варианта: GAT во втором разделе и GATv2 – в третьем.

Реализация слоя внимания графа в NumPy

Как уже говорилось ранее, нейронные сети работают на основе матричного умножения. Поэтому нам нужно преобразовать наши отдельные эмбеддинги в операции для всего графа. В этом разделе мы реализуем исходный слой внимания графа с нуля, чтобы правильно понимать внутреннюю работу механизма самовнимания. Естественно, этот процесс можно повторить несколько раз, чтобы создать многоголовное внимание.

Первый шаг состоит в том, чтобы исходный оператор внимания графа представить в матричном виде. Вот как мы определили его в последнем разделе:

$$h_i = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} x_j.$$

Отталкиваясь от линейного слоя графа, мы можем написать следующую формулу:

$$H = \tilde{A}^T W_\alpha X \mathbf{W}^T,$$

где W_α – это матрица, которая хранит значения α_{ij}.

В этом примере мы используем граф из предыдущей главы:

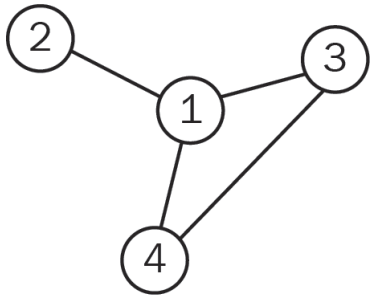


Рис. 7.3 ❖ Простой граф,
в котором узлы имеют разное количество соседей

Нам потребуется матрица смежности, включающая петли, и признаки узла. Давайте посмотрим, как это реализовать в NumPy.

1. Мы можем построить матрицу смежности на основе связей, показанных на рис. 7.3:

```
import numpy as np
np.random.seed(0)

A = np.array([
    [1, 1, 1, 1],
    [1, 1, 0, 0],
    [1, 0, 1, 1],
    [1, 0, 1, 1]
])
A
```

```
array([[1, 1, 1, 1],
       [1, 1, 0, 0],
       [1, 0, 1, 1],
       [1, 0, 1, 1]])
```

2. В качестве X мы сгенерируем случайную матрицу признаков узлов, используя функцию `np.random.uniform()`:

```
X = np.random.uniform(-1, 1, (4, 4))
X
```

```
array([[ 0.09762701,  0.43037873,  0.20552675,  0.08976637],
       [-0.1526904 ,  0.29178823, -0.12482558,  0.783546  ],
       [ 0.92732552, -0.23311696,  0.58345008,  0.05778984],
       [ 0.13608912,  0.85119328, -0.85792788, -0.8257414 ]])
```

3. Следующий шаг – определение матриц весов. На самом деле в слоях внимания графа будут две матрицы весов: обычная матрица весов **W** и матрица весов внимания W_{att}. Существуют разные способы их инициализации (например, инициализация Ксавьера или Хе), но в этом примере мы можем просто повторно воспользоваться функцией `np.random.uniform()`.

   Матрица **W** должна иметь размерность (*количество скрытых измерений*, *количество узлов*). Обратите внимание, что *количество узлов*=4 уже является зафиксированным, потому что оно представляет количество узлов в X. А *количество скрытых измерений*, наоборот, является произвольным: мы выбрали 2 в этом примере:

```
W = np.random.uniform(-1, 1, (2, 4))
W
```

```
array([[-0.95956321,  0.66523969,  0.5563135 ,  0.7400243 ],
       [ 0.95723668,  0.59831713, -0.07704128,  0.56105835]])
```

4. Матрица внимания применяется к конкатенации скрытых векторов. Ее размерность должна быть $(1, dim\_h \times 2)$:

```
W_att = np.random.uniform(-1, 1, (1, 4))
W_att
```

```
array([[-0.76345115,  0.27984204, -0.71329343,  0.88933783]])
```

5. Нам нужно сконкатенировать скрытые векторы узлов-источников и узлов-назначений. Простой способ получить пары узлов-источников и узлов-назначений – это посмотреть на матрицу смежности $\tilde{A}$ в формате COO: строки хранят узлы-источники, а столбцы – узлы-назначения. NumPy предлагает быстро и эффективно сделать это с помощью функции `np.where()`:

```
connections = np.where(A > 0)
connections
```

```
(array([0, 0, 0, 0, 1, 1, 2, 2, 2, 3, 3, 3], dtype=int64),
 array([0, 1, 2, 3, 0, 1, 0, 2, 3, 0, 2, 3], dtype=int64))
```

6. Мы можем сконкатенировать скрытые векторы узлов-источников и узлов-назначений с помощью функции `np.concatenate()`:

```
np.concatenate(
    [
        (X @ W.T)[connections[0]],
        (X @ W.T)[connections[1]]
    ], axis=1)
```

```
array([[ 0.37339233,  0.38548525,  0.37339233,  0.38548525],
       [ 0.37339233,  0.38548525,  0.85102612,  0.47765279],
       [ 0.37339233,  0.38548525, -0.67755906,  0.73566587],
       [ 0.37339233,  0.38548525, -0.65268413,  0.24235977],
       [ 0.85102612,  0.47765279,  0.37339233,  0.38548525],
       [ 0.85102612,  0.47765279,  0.85102612,  0.47765279],
       [-0.67755906,  0.73566587,  0.37339233,  0.38548525],
       [-0.67755906,  0.73566587, -0.67755906,  0.73566587],
       [-0.67755906,  0.73566587, -0.65268413,  0.24235977],
       [-0.65268413,  0.24235977,  0.37339233,  0.38548525],
       [-0.65268413,  0.24235977, -0.67755906,  0.73566587],
       [-0.65268413,  0.24235977, -0.65268413,  0.24235977]])
```

7. Затем применяем к этому результату линейное преобразование с помощью матрицы весов внимания W_{att}:

```
a = W_att @ np.concatenate(
    [
        (X @ W.T)[connections[0]],
        (X @ W.T)[connections[1]]
    ], axis=1).T
a
```

```
array([[-0.1007035 , -0.35942847,  0.96036209,  0.50390318, -0.43956122,
        -0.69828618,  0.79964181,  1.8607074 ,  1.40424849,  0.64260322,
         1.70366881,  1.2472099 ]])
```

8. Второй шаг состоит в том, чтобы применить функцию Leaky ReLU к предыдущему результату:

```
def leaky_relu(x, alpha=0.2):
    return np.maximum(alpha*x, x)

e = leaky_relu(a)
e

array([[-0.0201407 , -0.07188569,  0.96036209,  0.50390318, -0.08791224,
        -0.13965724,  0.79964181,  1.8607074 ,  1.40424849,  0.64260322,
         1.70366881,  1.2472099 ]])
```

9. У нас есть правильные значения, но нам нужно корректно разместить их в матрице. Эта матрица должна выглядеть как $\tilde{A}$, потому что, когда между двумя узлами нет связи, нет необходимости в ненормализованных оценках внимания. Чтобы построить такую матрицу, мы берем информацию об источниках и назначениях из объекта `connections`:

```
E = np.zeros(A.shape)
E[connections[0], connections[1]] = e[0]
E

array([[-0.0201407 , -0.07188569,  0.96036209,  0.50390318],
       [-0.08791224, -0.13965724,  0.        ,  0.        ],
       [ 0.79964181,  0.        ,  1.8607074 ,  1.40424849],
       [ 0.64260322,  0.        ,  1.70366881,  1.2472099 ]])
```

Итак, первое значение в `e`, равное –0.0201407, соответствует E_{00}, второе значение в `e`, равное –0.07188569, соответствует E_{01}, а седьмое значение 0.79964181 соответствует E_{20}, а не E_{12}.

10. Следующий шаг заключается в том, чтобы нормализовать каждую строку с оценками внимания. Мы напишем собственную функцию softmax для получения итоговых оценок внимания:

```
def softmax2D(x, axis):
    e = np.exp(x - np.expand_dims(np.max(x, axis=axis), axis))
    sum = np.expand_dims(np.sum(e, axis=axis), axis)
    return e / sum

W_alpha = softmax2D(E, 1)
W_alpha

array([[0.15862414, 0.15062488, 0.42285965, 0.26789133],
       [0.24193418, 0.22973368, 0.26416607, 0.26416607],
       [0.16208847, 0.07285714, 0.46834625, 0.29670814],
       [0.16010498, 0.08420266, 0.46261506, 0.2930773 ]])
```

11. С помощью этой матрицы внимания W_α мы получаем веса для каждого возможного соединения нейронной сети. Можем воспользоваться ею для вычисления нашей матрицы эмбеддингов, которая даст нам двумерные векторы для каждого узла:

```
H = A.T @ W_alpha @ X @ W.T
H

array([[-1.10126376,  1.99749693],
       [-0.33950544,  0.97045933],
       [-1.03570438,  1.53614075],
       [-1.03570438,  1.53614075]])
```

Наш слой внимания графа готов! Добавление многоголового внимания заключается в повторении этих шагов с разными матрицами $\mathbf{W}$ и W_{att} перед усреднением результатов.

Оператор внимания графа является важным строительным блоком для разработки GNN. В следующем разделе мы воспользуемся реализацией PyG для создания GAT.

Реализация GAT в PyTorch Geometric

Теперь у нас есть полное представление о том, как работает слой внимания графа. Эти слои мы можем скомбинировать и получить новую архитектуру – GAT. В этом разделе для реализации собственной модели с помощью PyG мы будем следовать рекомендациям статьи, посвященной исходному оператору внимания графа. Воспользуемся моделью для классификации узлов в наборах данных `Cora` и `CiteSeer`. В заключение мы прокомментируем эти результаты и сравним их.

Начнем с набора данных `Cora`.

1. Мы импортируем набор данных `Cora` с помощью класса `Planetoid` библиотеки PyTorch Geometric:

```
from torch_geometric.datasets import Planetoid

# импортируем набор данных из PyTorch Geometric
dataset = Planetoid(root=".", name="Cora")
data = dataset[0]
```

2. Импортируем необходимые библиотеки для создания собственного класса `GAT`, используя GATv2-слой:

```
import torch
torch.manual_seed(1)
import torch.nn.functional as F
from torch_geometric.nn import GATv2Conv, GCNConv
from torch.nn import Linear, Dropout
```

3. Пишем функцию `accuracy()` для оценки качества модели:

```
def accuracy(y_pred, y_true):
    """Вычисляет правильность."""
    return torch.sum(y_pred == y_true) / len(y_true)
```

4. Теперь мы инициализируем класс с двумя улучшенными слоями внимания графа. Обратите внимание, что важно указать количество голов, используемых для многоголового (множественного) внимания. Авторы в своей статье заявили, что восемь голов улучшили качество работы первого слоя, но для второго это не имело никакого значения:

```
class GAT(torch.nn.Module):
    def __init__(self, dim_in, dim_h, dim_out, heads=8):
        super().__init__()
        self.gat1 = GATv2Conv(dim_in, dim_h, heads=heads)
        self.gat2 = GATv2Conv(dim_h*heads, dim_out, heads=1)
```

5. Если сравнивать с реализацией GCN, то здесь мы добавляем два слоя дропаута, чтобы предотвратить переобучение. Они случайным образом обнуляют некоторые значения входного тензора с заранее определенной вероятностью (в данном случае с вероятностью 0.6). В соответствии с исходной статьей мы также используем функцию Exponential Linear Unit (ELU), которая является экспоненциальной версией Leaky ReLU:

```
    def forward(self, x, edge_index):
        h = F.dropout(x, p=0.6, training=self.training)
        h = self.gat1(h, edge_index)
        h = F.elu(h)
        h = F.dropout(h, p=0.6, training=self.training)
        h = self.gat2(h, edge_index)
        return F.log_softmax(h, dim=1)
```

6. Метод `.fit()` идентичен методу `.fit()` в реализации GCN. По словам авторов, параметры оптимизатора Adam были настроены так, чтобы наилучшим образом соответствовать набору данных `Cora`:

```
    def fit(self, data, epochs):
        criterion = torch.nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(self.parameters(), lr=0.01,
                                     weight_decay=0.01)

        self.train()
        for epoch in range(epochs+1):
            optimizer.zero_grad()
            out = self(data.x, data.edge_index)
            loss = criterion(out[data.train_mask],
                             data.y[data.train_mask])
            acc = accuracy(out[data.train_mask].argmax(dim=1),
                           data.y[data.train_mask])
            loss.backward()
            optimizer.step()

            if(epoch % 20 == 0):
                val_loss = criterion(out[data.val_mask],
                                     data.y[data.val_mask])
                val_acc = accuracy(out[data.val_mask].argmax(dim=1),
                                   data.y[data.val_mask])
```

```
            print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: '
                  f'{loss:.3f} | Правильность на обуч. выборке: '
                  f'{acc*100:>5.2f}% \n| Функция потерь на валид. '
                  f'выборке: {val_loss:.2f} | Правильность на валид. '
                  f'выборке: {val_acc*100:.2f}%')
```

7. Метод `.test()` идентичен методу `.test()` ранее написанных классов:

```
@torch.no_grad()
def test(self, data):
    self.eval()
    out = self(data.x, data.edge_index)
    acc = accuracy(out.argmax(dim=1)[data.test_mask],
                   data.y[data.test_mask])
    return acc
```

8. Мы создаем экземпляр класса `GAT` и обучаем модель, задав 100 эпох:

```
# создаем модель GAT
gat = GAT(dataset.num_features, 32, dataset.num_classes)
print(gat)

# обучаем
gat.fit(data, epochs=100)
```

```
GAT(
  (gat1): GATv2Conv(1433, 32, heads=8)
  (gat2): GATv2Conv(256, 7, heads=1)
)
Эпоха   0:
| Функция потерь на обуч. выборке: 1.969 | Правильность на обуч. выборке: 15.00%
| Функция потерь на валид. выборке: 1.96 | Правильность на валид. выборке: 11.80%
Эпоха  20:
| Функция потерь на обуч. выборке: 0.259 | Правильность на обуч. выборке: 96.43%
| Функция потерь на валид. выборке: 1.10 | Правильность на валид. выборке: 67.60%
Эпоха  40:
| Функция потерь на обуч. выборке: 0.163 | Правильность на обуч. выборке: 98.57%
| Функция потерь на валид. выборке: 0.90 | Правильность на валид. выборке: 70.80%
Эпоха  60:
| Функция потерь на обуч. выборке: 0.205 | Правильность на обуч. выборке: 98.57%
| Функция потерь на валид. выборке: 0.96 | Правильность на валид. выборке: 69.00%
Эпоха  80:
| Функция потерь на обуч. выборке: 0.130 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 0.91 | Правильность на валид. выборке: 70.80%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.148 | Правильность на обуч. выборке: 99.29%
| Функция потерь на валид. выборке: 0.90 | Правильность на валид. выборке: 73.00%
```

9. Теперь оценим качество построенной модели:

```
# тестируем
acc = gat.test(data)
print(f'Правильность GAT на тесте (Cora): {acc*100:.2f}%')
```

```
Правильность GAT на тесте (Cora): 82.00%
```

Оценка правильности немного лучше оценки правильности, которую мы получили с помощью GCN. Теперь оценим качество модели после применения архитектуры GAT на втором наборе данных.

Мы воспользуемся новым популярным набором данных для классификации узлов под названием `CiteSeer` (лицензия MIT). Как и набор данных `Cora`, он представляет собой сеть исследовательских статей, в которой каждое соединение является цитатой. CiteSeer включает в себя 3327 узлов, характеристики которых отражают наличие (1) или отсутствие (0) 3703 слов в статье. Задача, поставленная применительно к этому набору данных, – это правильная классификация этих узлов по шести классам. На рис. 7.4 приведена визуализация `CiteSeer`, созданная с помощью yEd Live:

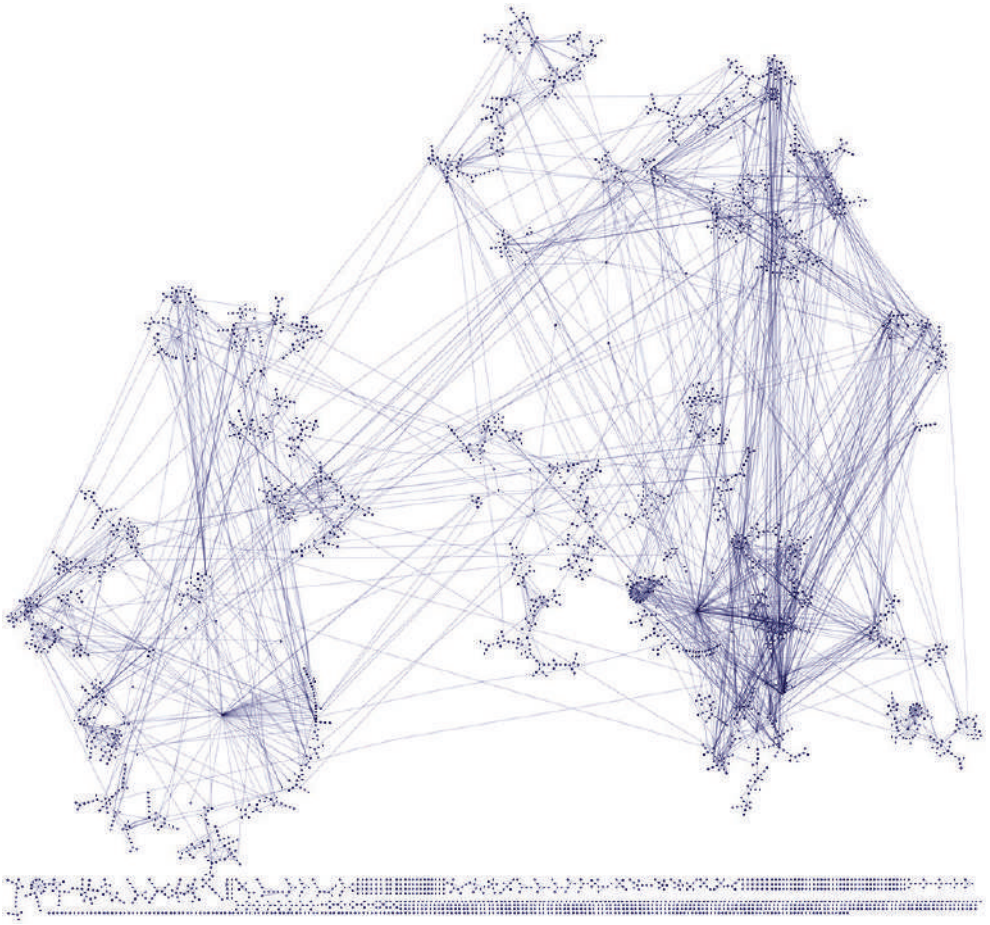


Рис. 7.4 ❖ Набор данных CiteSeer, визуализированный с помощью yEd Live

По сравнению с `Cora` этот набор данных больше по количеству узлов (количество узлов возросло с 2708 до 3327), а также по размерности признакового

пространства (количество признаков возросло с 1433 до 3703). Однако к нему можно применить ту же самую модель GAT.

1. Сначала загружаем набор `CiteSeer`:

```
dataset = Planetoid(root=".", name="CiteSeer")
data = dataset[0]
data

Data(x=[3327, 3703], edge_index=[2, 9104], y=[3327], train_mask=[3327], val_
mask=[3327], test_mask=[3327])
```

2. Теперь визуализируем распределение степеней узлов, используя программный код из предыдущей главы:

```
import matplotlib.pyplot as plt
from torch_geometric.utils import degree
from collections import Counter

# создаем список степеней
degrees = degree(dataset[0].edge_index[0]).numpy()

# вычисляем количество узлов для каждой степени
numbers = Counter(degrees)

# столбиковая диаграмма
fig, ax = plt.subplots()
ax.set_xlabel('Степень узла')
ax.set_ylabel('Количество узлов')
plt.bar(numbers.keys(), numbers.values())
```

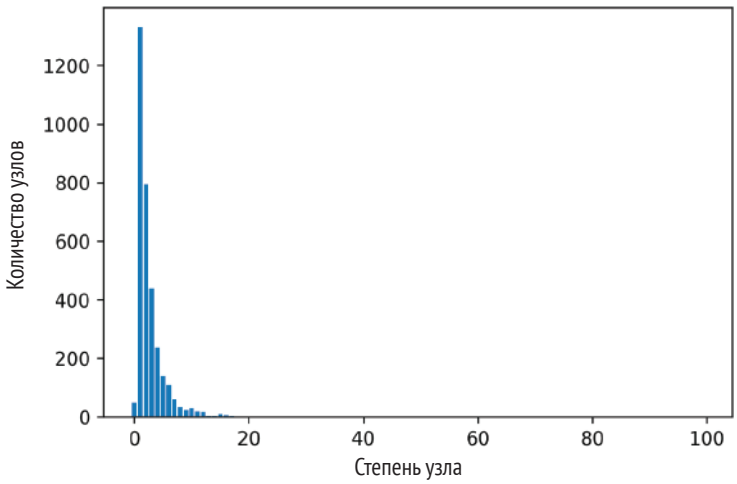


Рис. 7.5 ❖ Распределение степеней узлов в наборе данных CiteSeer

Перед нами типичное распределение с длинным хвостом, но с одной особенностью: некоторые узлы имеют нулевую степень! Другими сло-

вами, эти узлы не соединены ни с какими другими узлами. Можно предположить, что их будет гораздо сложнее классифицировать, чем остальные.

3. Мы создаем экземпляр класса `GAT` и обучаем модель, задав 100 эпох:

```
# создаем модель GAT
gat = GAT(dataset.num_features, 16, dataset.num_classes)
print(gat)

# обучаем
gat.fit(data, epochs=100)
```

```
GAT(
  (gat1): GATv2Conv(3703, 16, heads=8)
  (gat2): GATv2Conv(128, 6, heads=1)
)
Эпоха   0:
| Функция потерь на обуч. выборке: 1.803 | Правильность на обуч. выборке: 15.83%
| Функция потерь на валид. выборке: 1.81 | Правильность на валид. выборке: 16.60%
Эпоха  20:
| Функция потерь на обуч. выборке: 0.175 | Правильность на обуч. выборке: 96.67%
| Функция потерь на валид. выборке: 1.15 | Правильность на валид. выборке: 60.40%
Эпоха  40:
| Функция потерь на обуч. выборке: 0.134 | Правильность на обуч. выборке: 97.50%
| Функция потерь на валид. выборке: 1.22 | Правильность на валид. выборке: 62.00%
Эпоха  60:
| Функция потерь на обуч. выборке: 0.135 | Правильность на обуч. выборке: 97.50%
| Функция потерь на валид. выборке: 1.21 | Правильность на валид. выборке: 61.60%
Эпоха  80:
| Функция потерь на обуч. выборке: 0.111 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 1.15 | Правильность на валид. выборке: 63.60%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.099 | Правильность на обуч. выборке: 100.00%
| Функция потерь на валид. выборке: 1.24 | Правильность на валид. выборке: 58.60%
```

4. Теперь оценим качество построенной модели:

```
# тестируем
acc = gat.test(data)
print(f'Правильность GAT на тесте (CiteSeer): {acc*100:.2f}%')
```

```
Правильность GAT на тесте (CiteSeer): 67.40%
```

Это хороший результат? На этот раз нам не с чем сравнивать.

В статье «Pitfalls of Graph Neural Network Evaluation» Шур совместно с коллегами (Schur et al., 2018) утверждал, что GAT немного лучше, чем GCN на наборе `Cora` (82.8 ± 0.6 % против 81.9 ± 0.8 %) и наборе `CiteSeer` (71.0 ± 0.6 % против 69.5 ± 0.9 %). Авторы также отмечают, что оценки правильности не подчиняются нормальному распределению, что делает использование стандартного отклонения менее релевантным. При проведении такого рода тестов важно помнить об этом.

Ранее я предположил, что изолированные узлы могут отрицательно повлиять на качество работы модели. Мы можем проверить эту гипотезу, построив оценки правильности для каждой степени узла.

Получаем результаты модели:

```
# получаем результаты
out = gat(data.x, data.edge_index)
```

1. Получаем список степеней:

```
# создаем список степеней
degrees = degree(data.edge_index[0]).numpy()
```

2. Создаем пустые списки для хранения оценок правильности и размеров степеней:

```
# здесь храним оценки правильности
# и размеры степеней
accuracies = []
sizes = []
```

3. Получаем оценки правильности для узлов со степенью от 0 до 5:

```
# правильность для степеней в диапазоне от 0 до 5
for i in range(0, 6):
    mask = np.where(degrees == i)[0]
    accuracies.append(accuracy(out.argmax(dim=1)[mask],
                               data.y[mask]))
    sizes.append(len(mask))
```

4. Мы повторяем этот процесс для узлов со степенью выше 5:

```
# правильность для степеней > 5
mask = np.where(degrees > 5)[0]
accuracies.append(accuracy(out.argmax(dim=1)[mask],
                           data.y[mask]))
sizes.append(len(mask))
```

5. Визуализируем эти оценки правильности:

```
# столбиковая диаграмма
fig, ax = plt.subplots()
ax.set_xlabel('Степень узла')
ax.set_ylabel('Оценка правильности')
plt.bar(['0','1','2','3','4','5','6+'], accuracies)
for i in range(0, 7):
    plt.text(i, accuracies[i], f'{accuracies[i]*100:.2f}%',
             ha='center', color='black')
for i in range(0, 7):
    plt.text(i, accuracies[i]//2, sizes[i],
             ha='center', color='white')
```

Рисунок 7.6 подтверждает нашу гипотезу: узлы с небольшим количеством соседей сложнее правильно классифицировать. Более того, это показывает,

что, как правило, чем выше степень узла, тем выше оценка правильности. Это вполне естественно, поскольку большее количество соседей предоставит GNN больше информации для прогнозирования.

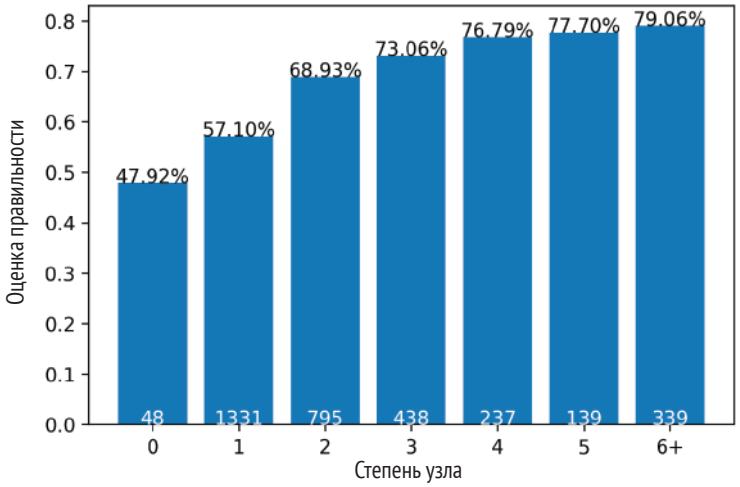


Рис. 7.6 ❖ Оценки правильности для определенной степени узла (CiteSeer)

Выводы

В этой главе мы познакомились с новой важной архитектурой – GAT. Мы рассмотрели в четыре этапа, как эта архитектура устроена внутри: от линейной трансформации до многоголового внимания. Мы увидели, как она работает на практике, реализовав слой внимания графа в NumPy. Наконец, применили модель GAT (используя GATv2) к наборам данных `Cora` и `CiteSeer` и получили хорошие оценки правильности. Мы показали, что оценки правильности зависят от количества соседей, что является первым шагом на пути к анализу ошибок.

В главе 8 «Масштабирование графовых нейронных сетей с помощью GraphSAGE» мы представим новую архитектуру, предназначенную для управления большими графами. Чтобы проверить это утверждение, реализуем ее на новом наборе данных, в несколько раз большем, чем те, которые видели до сих пор. Мы поговорим о двух различных подходах к машинному обучению на графах – трансдуктивном и индуктивном обучении, которые используют практикующие специалисты по GNN.

ЧАСТЬ III

ПРОДВИНУТЫЕ МЕТОДЫ

В третьей части книги мы углубимся в более продвинутые и специализированные архитектуры GNN, которые были разработаны для решения различных задач, связанных с графами. Мы рассмотрим современные модели GNN, разработанные для конкретных задач и областей, которые могут более эффективно решать существующие проблемы и отвечать современным требованиям. Кроме того, мы представим обзор нескольких новых задач на основе графов, которые можно решить с помощью GNN, таких как прогнозирование связей и классификация графов, и продемонстрируем их применение на практических примерах.

К концу этой части вы сможете понять и реализовать передовые архитектуры GNN и применять их для решения собственных задач на основе графов. У вас будет полное понимание специализированных GNN и их преимуществ, а также практический опыт работы. Эти знания позволят вам применить GNN в реальных задачах и, возможно, внести вклад в разработку новых и инновационных архитектур GNN.

Эта часть состоит из следующих глав:

- глава 8 «Масштабирование графовых нейронных сетей с помощью GraphSAGE»,
- глава 9 «Определение выразительности для классификации графов»,
- глава 10 «Прогнозирование связей с помощью графовых нейронных сетей»,
- глава 11 «Генерация графов с помощью графовых нейронных сетей»,
- глава 12 «Обучение на гетерогенных графах»,
- глава 13 «Темпоральные графовые нейронные сети»,
- глава 14 «Интерпретация графовых нейронных сетей».

Глава 8

Масштабирование графовых нейронных сетей с помощью GraphSAGE

GraphSAGE – это архитектура GNN, предназначенная для обработки больших графов. В технологической отрасли **масштабируемость** (scalability) является ключевым фактором успеха. Современные системы обработки данных по своей сути должны обслуживать миллионы пользователей. Это требует фундаментального изменения работы модели GNN по сравнению с GCN и GAT. Поэтому неудивительно, что GraphSAGE является предпочтительной архитектурой для таких технологических компаний, как Uber Eats и Pinterest.

В этой главе мы узнаем о двух основных идеях GraphSAGE. Сначала опишем метод **семплирования соседей** (neighbor sampling), который лежит в основе эффективности GraphSAGE с точки зрения масштабируемости. Затем рассмотрим три оператора агрегации, которые используются для создания эмбеддингов узлов. Помимо оригинального подхода, мы также подробно опишем варианты, предложенные Uber Eats и Pinterest.

Кроме того, GraphSAGE предлагает новые возможности в плане обучения. Мы реализуем два способа обучения GNN для двух задач: классификации узлов на основе набора `PubMed` и **многометочной классификации** (multi-label classification) для **белок-белковых взаимодействий** (protein-protein interactions). Наконец, обсудим преимущества нового **индуктивного** (inductive) подхода и способы его использования.

К концу этой главы вы поймете, как и почему работает алгоритм семплирования соседей. Вы сможете реализовать его для создания мини-батчей и ускорить обучение на большинстве архитектур GNN с использованием GPU. Кроме того, вы освоите индуктивное обучение и многометочную классификацию на графах.

В этой главе будут рассмотрены следующие темы:

- «Знакомство с GraphSAGE»;
- «Классификация узлов на примере набора PubMed»;
- «Индуктивное обучение на белок-белковых взаимодействиях».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter08.

Знакомство с GraphSAGE

Гамильтон совместо с коллегами представил GraphSAGE в 2017 году (см. пункт [1] раздела «Дополнительное чтение») как фреймворк для индуктивного обучения представлениям на больших графах (с более чем 100 000 узлов). Его цель – генерировать эмбеддинги узлов для последующих задач, например для классификации узлов. Кроме того, он решает две проблемы, связанные с GCN и GAT, – масштабирование для больших графов и способность эффективно обобщать на новые данные. В этом разделе мы объясним, как реализовать GraphSAGE, описав два основных компонента:

- семплирование соседей;
- агрегацию.

Давайте разберем каждый компонент.

Семплирование соседей

До сих пор мы не обсуждали важную концепцию традиционных нейронных сетей – **мини-батчинг** (mini-batching). Мини-батчинг подразумевает разбиение нашего набора данных на более мелкие подмножества, называемые батчами или пакетами. Они используются в **градиентном спуске** (gradient descent) – алгоритме оптимизации, который находит лучшие веса и смещения во время обучения. Существует три типа градиентного спуска.

- **Пакетный градиентный спуск** (batch gradient descent): веса и смещения обновляются после обработки всего обучающего набора (каждую эпоху). Именно этот метод мы и реализовывали до сих пор. Однако это медленный процесс, требующий размещения всего набора данных в памяти.
- **Стохастический градиентный спуск** (stochastic gradient descent): веса и смещения обновляются после обработки каждого отдельного

примера обучающего набора. Он характеризуется зашумленностью, поскольку ошибки не усредняются. Однако его можно использовать для онлайн-обучения.

- **Мини-пакетный градиентный спуск**, или **градиентный спуск на мини-батчах** (mini-batch gradient descent): веса и смещения обновляются после обработки небольшого мини-батча из *n* обучающих примеров. Этот метод быстрее (мини-батчи могут обрабатываться параллельно с использованием GPU) и приводит к более стабильной сходимости. Кроме того, набор данных может превышать доступную память, что важно для обработки больших графов.

На практике мы используем более продвинутые оптимизаторы, такие как **RMSprop** или Adam, в которых также реализован мини-батчинг.

Разбиение табличного набора данных является несложной процедурой: такой набор просто состоит из *n* наблюдений (строк). Однако разбиение становится проблемой для графовых наборов данных графа: как нам отобрать *n* узлов, не разрушив важные связи? Если мы не проявим осторожность, то можем получить коллекцию изолированных узлов, когда невозможно выполнить какую-либо агрегацию.

Нам нужно подумать о том, как графовые нейронные сети (GNN) используют наборы данных. Каждый GNN-слой вычисляет эмбеддинги узлов на основе их соседей. Это означает, что для вычисления эмбеддинга требуются только непосредственные соседи этого узла (**1-й уровень**). Если у нашей графовой нейронной сети два GNN-слоя, нам нужны эти соседи и их собственные соседи (**2-й уровень**) и т. д. (см. рис. 8.1). Остальная часть сети не имеет значения для вычисления этих эмбеддингов узлов.

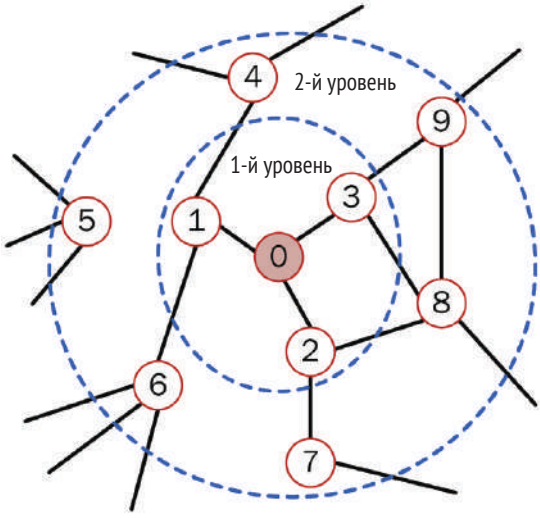


Рис. 8.1 ❖ Граф с узлом 0 в качестве целевого узла и соседями 1-го и 2-го уровней

Этот прием позволяет использовать в мини-батчах графы вычислений, которые описывают всю последовательность операций для вычисления эмбеддинга узла. На рис. 8.2 показано интуитивное представление графа вычислений для узла 0.

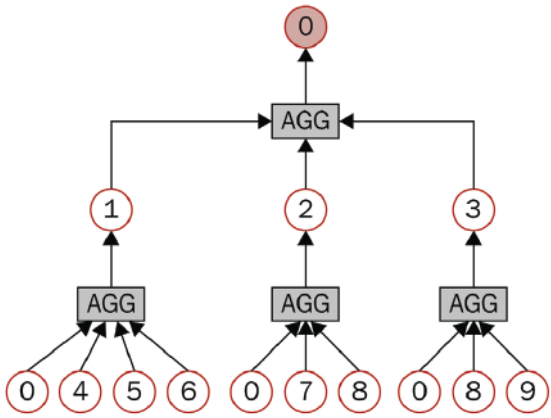


Рис. 8.2 ❖ Граф вычислений для узла 0

Нам нужно выполнить агрегацию узлов-соседей на втором уровне, чтобы вычислить эмбеддинг узлов-соседей, находящихся на первом уровне. Эти эмбеддинги затем агрегируются для получения эмбеддинга узла 0. Однако в этой схеме есть две проблемы:

- граф вычислений становится экспоненциально большим в зависимости от количества уровней;
- узлы с очень высокой степенью связности (например, знаменитости в социальной сети), называемые **узлами-концентраторами** или **узлами-хабами** (hub nodes), создают огромные графы вычислений.

Чтобы решить эти проблемы, необходимо ограничить размер наших вычислительных графов. Авторы GraphSAGE предлагают метод под названием семплирование соседей. Вместо добавления каждого соседа в граф вычислений мы отбираем заранее определенное количество соседей. Например, мы решили сохранить (не более) трех соседей на первом уровне и пяти соседей на втором уровне. Следовательно, граф вычислений в этом случае не может превышать $3 \times 5 = 15$ узлов.

Низкая частота семплирования (т. е. когда отбираем маленькое количество соседей) более эффективна, но делает обучение более случайным (дает более высокую дисперсию). Кроме того, количество GNN-слоев (уровней) должно оставаться низким, чтобы избежать экспоненциально больших графов вычислений. Семплирование соседей позволяет обрабатывать большие графы, но при этом приходится сокращать важную информацию, что может негативно повлиять на качество модели, например на правильность классификации. Обратите внимание, что графы вычислений включают в себя

множество избыточных вычислений, что делает весь процесс вычислений менее эффективным.

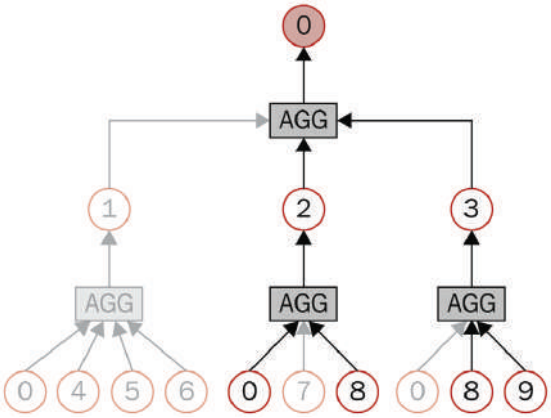


Рис. 8.3 ❖ Граф вычислений с семплированием соседей, сохраняем двух соседей 1-го уровня и двух соседей 2-го уровня

Однако это случайное семплирование не единственный метод, который мы можем использовать. У Pinterest есть собственная версия GraphSAGE под названием PinSAGE, обеспечивающая работу рекомендательной системы (см. [2] раздела «Дополнительное чтение»). В ней реализован другой вариант семплирования с использованием случайных блужданий. В PinSAGE сохранена идея фиксированного числа соседей, но реализованы случайные блуждания, чтобы посмотреть, какие узлы встречаются чаще всего. Эта частота встречаемости узлов определяет их относительную важность. Стратегия семплирования PinSAGE позволяет отбирать наиболее важные узлы и на практике является более эффективной.

Агрегация

Итак, мы выяснили, как выбирать соседние узлы, но нам по-прежнему нужно получить эмбеддинги. Эту операцию выполняет оператор агрегации (или агрегатор). В GraphSAGE авторы предложили три решения:

- агрегатор на основе среднего (или агрегатор на основе усреднения),
- LSTM-агрегатор, или агрегатор на основе **долгой краткосрочной памяти** (long short-term memory),
- агрегатор на основе пулинга.

Мы подробнее рассмотрим агрегатор на основе среднего, поскольку его проще всего понять. Во-первых, агрегатор на основе среднего берет эмбеддинги целевых узлов и их отобранных соседей, чтобы усреднить их. Затем к этому результату применяется линейное преобразование с матрицей весов **W**.

Агрегатор на основе среднего можно проиллюстрировать следующей формулой, где σ – нелинейная функция типа ReLU или tanh:

$$h_i' = \sigma(\mathbf{W} \cdot mean_{j \in \mathcal{N}_i}(h_j)).$$

В случае реализации GraphSAGE в PyG и Uber Eats [3] мы используем две матрицы весов вместо одной, первая предназначена для целевого узла, а вторая – для соседних узлов. Этот агрегатор можно записать следующим образом:

$$h_i' = \sigma(\mathbf{W}_1 h_i + \mathbf{W}_2 \cdot mean_{j \in \mathcal{N}_i}(h_j)).$$

Агрегатор на основе долгой краткосрочной памяти (LSTM-агрегатор) основан на архитектуре LSTM, популярном типе рекуррентных нейронных сетей. В сравнении с агрегатором на основе среднего LSTM-агрегатор теоретически способен различать более сложные структуры графов и, следовательно, создавать более качественные эмбеддинги. Проблема заключается в том, что рекуррентные нейронные сети рассматривают только последовательности входов типа предложения с началом и концом. Однако узлы не имеют никакой последовательности. Поэтому мы выполняем случайные перестановки соседей узла, чтобы решить эту проблему. Это решение позволяет использовать LSTM-архитектуру, не полагаясь на какую-либо последовательность входов.

Наконец, агрегатор на основе пулинга работает в два этапа. Во-первых, эмбеддинг каждого соседа передается в MLP для создания нового вектора. Во-вторых, выполняется поэлементная операция максимума, чтобы сохранить только наивысшее значение для каждого признака.

Мы не ограничены этими тремя вариантами и можем реализовать другие агрегаторы в рамках структуры GraphSAGE. На самом деле главная идея GraphSAGE заключается в эффективном семплировании соседей. В следующем разделе мы продемонстрируем его для классификации узлов на новом наборе данных.

Классификация узлов на примере набора данных PubMed

В этом разделе мы реализуем архитектуру GraphSAGE для выполнения классификации узлов на наборе данных `PubMed` (доступно по лицензии MIT на https://github.com/kimiyoung/planetoid) [4].

Ранее мы рассмотрели два набора данных на основе сети цитирования из того же семейства Planetoid – `Cora` и `CiteSeer`. Набор данных `PubMed` представляет собой схожий, но более крупный граф, состоящий из 19 717 узлов и 88 648 ребер. На рис. 8.4 показана визуализация этого набора данных, созданного с помощью Gephi (https://gephi.org/).

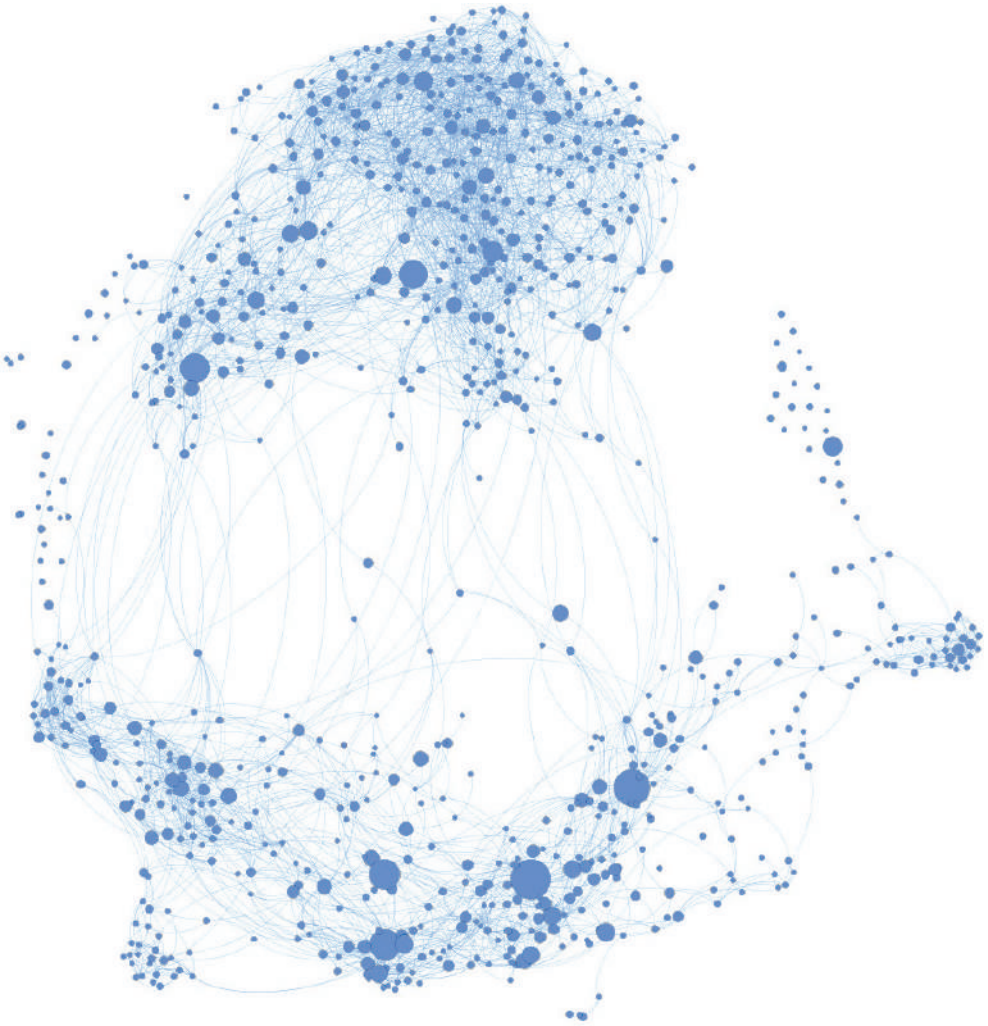


Рис. 8.4 ❖ Визуализация набора данных PubMed

Характеристики узлов представлены векторами слов, взвешенными по TF-IDF, размерности 500. Цель – правильно классифицировать узлы на три категории: экспериментальный диабет, диабет 1-го типа и диабет 2-го типа. Давайте реализуем GraphSAGE пошагово с использованием PyG.

1. Зададим некоторые настройки и напишем собственную функцию для обеспечения воспроизводимости результатов:

```
import torch
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

def set_seed():
    """
```

```
    Задает стартовое значение генератора псевдослучайных
    чисел для воспроизводимости.
    """
    torch.manual_seed(-1)
    torch.cuda.manual_seed(0)
    torch.cuda.manual_seed_all(0)
```

2. Мы загружаем набор данных PubMed с помощью класса Planetoid и выводим некоторую информацию о графе:

```
from torch_geometric.datasets import Planetoid

dataset = Planetoid(root='.', name="Pubmed")
data = dataset[0]

# печатаем информацию о наборе данных
print(f'Набор данных: {dataset}')
print('---------------')
print(f'Количество графов: {len(dataset)}')
print(f'Количество узлов: {data.x.shape[0]}')
print(f'Количество признаков: {dataset.num_features}')
print(f'Количество классов: {dataset.num_classes}')

# печатаем информацию о графе
print(f'\nГраф:')
print('------')
print(f'Узлы для обучения: {sum(data.train_mask).item()}')
print(f'Узлы для валидации: {sum(data.val_mask).item()}')
print(f'Узлы для тестирования: {sum(data.test_mask).item()}')
print(f'Ребра являются ориентированными: {data.is_directed()}')
print(f'У графа есть изолированные узлы: {data.has_isolated_nodes()}')
print(f'У графа есть петли: {data.has_self_loops()}')
```

3. Получаем вывод:

```
Набор данных: Pubmed()
---------------
Количество графов: 1
Количество узлов: 19717
Количество признаков: 500
Количество классов: 3

Граф:
------
Узлы для обучения: 60
Узлы для валидации: 500
Узлы для тестирования: 1000
Ребра являются ориентированными: False
У графа есть изолированные узлы: False
У графа есть петли: False
```

Как видите, на 1000 узлов для тестирования приходится всего 60 узлов для обучения, что довольно непривычно (разбиение 6/94). К счастью

для нас, PubMed, включающий 19 717 узлов, будет очень быстро обрабатываться с помощью GraphSAGE.

4. Первым шагом метода GraphSAGE является семплирование соседей. PyG реализует класс NeighborLoader для выполнения этого шага. Давайте установим 5 соседей для целевого узла и 10 соседей для каждого соседа целевого узла. Мы разобьем 60 целевых узлов (узлы для обучения) на батчи по 16 узлов, что должно привести к четырем батчам:

```
from torch_geometric.loader import NeighborLoader

# создаем батчи с семплированием соседей
train_loader = NeighborLoader(
    data,
    num_neighbors=[5, 10],
    batch_size=16,
    input_nodes=data.train_mask,
)
```

5. Давайте проверим, что мы получили четыре подграфа (батча):

```
# печатаем информацию о каждом подграфе
for i, subgraph in enumerate(train_loader):
    print(f'Подграф {i}: {subgraph}')
```

```
Подграф 0: Data(x=[419, 500], edge_index=[2, 464], y=[419], train_mask=[419], val_
mask=[419], test_mask=[419], input_id=[16], batch_size=16)
Подграф 1: Data(x=[268, 500], edge_index=[2, 311], y=[268], train_mask=[268], val_
mask=[268], test_mask=[268], input_id=[16], batch_size=16)
Подграф 2: Data(x=[289, 500], edge_index=[2, 324], y=[289], train_mask=[289], val_
mask=[289], test_mask=[289], input_id=[16], batch_size=16)
Подграф 3: Data(x=[189, 500], edge_index=[2, 225], y=[189], train_mask=[189], val_
mask=[189], test_mask=[189], input_id=[12], batch_size=12)
```

6. Эти подграфы содержат более 60 узлов, что нормально, поскольку семплированию можно подвергнуть любого соседа. Мы даже можем визуализировать их:

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt

# визуализируем каждый подграф
fig = plt.figure(figsize=(16,16))
for idx, (subdata, pos) in enumerate(zip(train_loader, [221, 222, 223, 224])):
    G = to_networkx(subdata, to_undirected=True)
    ax = fig.add_subplot(pos)
    ax.set_title(f'Подграф {idx}', fontsize=24)
    plt.axis('off')
    nx.draw_networkx(G,
                     pos=nx.spring_layout(G, seed=0),
```

```
                    with_labels=False,
                    node_color=subdata.y,
                    )
plt.show()
```

7. Получаем следующий график:

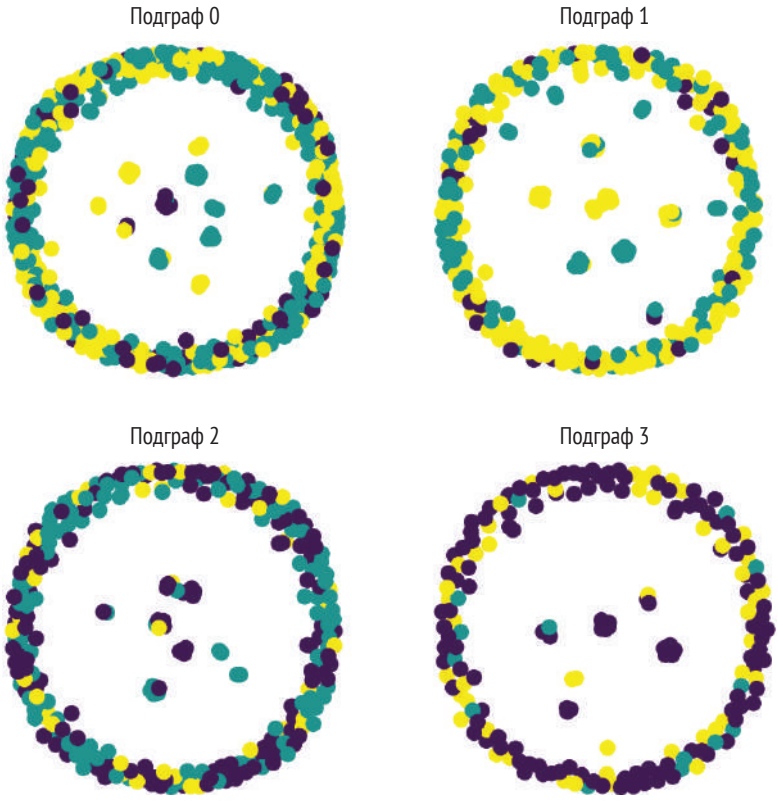


Рис. 8.5 ❖ Визуализация подграфов, полученных с помощью семплирования соседей

У большинства этих узлов – степень 1 из-за принципа работы семплирования соседей. В данном случае это не проблема, поскольку их эмбеддинги используются в графе вычислений только один раз для вычисления эмбеддингов второго слоя.

8. Теперь реализуем функцию, вычисляющую правильность:

```
def accuracy(pred_y, y):
    """Вычисляем правильность."""
    return ((pred_y == y).sum() / len(y)).item()
```

9. Давайте создадим класс `GraphSAGE`, используя два SAGEConv-слоя (по умолчанию выбран агрегатор на основе среднего):

```
import torch.nn.functional as F
from torch_geometric.nn import SAGEConv


class GraphSAGE(torch.nn.Module):
    """GraphSAGE"""
    def __init__(self, dim_in, dim_h, dim_out):
        super().__init__()
        self.sage1 = SAGEConv(dim_in, dim_h)
        self.sage2 = SAGEConv(dim_h, dim_out)
```

10. Эмбеддинги вычисляются с использованием двух агрегаторов на основе среднего. Кроме того, мы используем нелинейную функцию (ReLU) и слой дропаута:

```
    def forward(self, x, edge_index):
        h = self.sage1(x, edge_index)
        h = torch.relu(h)
        h = F.dropout(h, p=0.5, training=self.training)
        h = self.sage2(h, edge_index)
        return h
```

11. Теперь, когда нам нужно учитывать батчи, в методе `.fit()` надо итерировать сначала по эпохам, а затем по батчам. Измеряемые метрики повторно инициализируем каждую эпоху, при этом выводим метрики через каждые 20 эпох (метрику делим на количество батчей, представляющих эпоху):

```
    def fit(self, loader, epochs):
        criterion = torch.nn.CrossEntropyLoss()
        optimizer = torch.optim.Adam(self.parameters(), lr=0.01)

        self.train()
        for epoch in range(epochs+1):
            total_loss = 0
            acc = 0
            val_loss = 0
            val_acc = 0

            # обучаем на батчах
            for batch in loader:
                optimizer.zero_grad()
                out = self(batch.x, batch.edge_index)
                loss = criterion(out[batch.train_mask],
                                 batch.y[batch.train_mask])
                total_loss += loss.item()
                acc += accuracy(out[batch.train_mask].argmax(dim=1),
                                batch.y[batch.train_mask])
                loss.backward()
                optimizer.step()

                # валидируем
                val_loss += criterion(out[batch.val_mask],
```

```
                                    batch.y[batch.val_mask])
            val_acc += accuracy(out[batch.val_mask].argmax(dim=1),
                                batch.y[batch.val_mask])

        # печатаем функцию потерь и правильность каждые 20 итераций
        if epoch % 20 == 0:
            print(f'Эпоха {epoch:>3}:\n | Функция потерь на обуч. выборке: '
                  f'{loss/len(loader):.3f} | Правильность на обуч. выборке: '
                  f'{acc/len(loader)*100:>6.2f}%\n | Функция потерь на валид. '
                  f'выборке: {val_loss/len(train_loader):.2f}'
                  f'| Правильность на валид. выборке: '
                  f'{val_acc/len(train_loader)*100:.2f}%')
```

12. Метод `.test()` мы не меняем, поскольку не используем батчи для тестового набора:

```
@torch.no_grad()
def test(self, data):
    self.eval()
    out = self(data.x, data.edge_index)
    acc = accuracy(out.argmax(dim=1)[data.test_mask],
                   data.y[data.test_mask])
    return acc
```

13. Теперь создаем модель с размерностью скрытого слоя, равной 64, и обучаем, задав 200 эпох:

```
set_seed()

# создаем GraphSAGE
graphsage = GraphSAGE(dataset.num_features, 64, dataset.num_classes)
print(graphsage)

# обучаем
graphsage.fit(train_loader, 200)
```

```
GraphSAGE(
  (sage1): SAGEConv(500, 64, aggr=mean)
  (sage2): SAGEConv(64, 3, aggr=mean)
)
Эпоха   0:
 | Функция потерь на обуч. выборке: 0.314 | Правильность на обуч. выборке:  28.77%
 | Функция потерь на валид. выборке: 1.12| Правильность на валид. выборке: 24.94%
Эпоха  20:
 | Функция потерь на обуч. выборке: 0.002 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.67| Правильность на валид. выборке: 66.43%
Эпоха  40:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.52| Правильность на валид. выборке: 84.72%
Эпоха  60:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.64| Правильность на валид. выборке: 75.13%
```

```
Эпоха  80:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.45| Правильность на валид. выборке: 83.33%
Эпоха 100:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.47| Правильность на валид. выборке: 83.02%
Эпоха 120:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.50| Правильность на валид. выборке: 80.35%
Эпоха 140:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.44| Правильность на валид. выборке: 77.43%
Эпоха 160:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.62| Правильность на валид. выборке: 78.22%
Эпоха 180:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.33| Правильность на валид. выборке: 84.15%
Эпоха 200:
 | Функция потерь на обуч. выборке: 0.000 | Правильность на обуч. выборке: 100.00%
 | Функция потерь на валид. выборке: 0.74| Правильность на валид. выборке: 71.43%
```

14. Теперь оценим качество построенной модели:

```
# тестируем
acc = graphsage.test(data)
print(f'Правильность модели GraphSAGE на тесте: {acc*100:.2f}%')

Правильность модели GraphSAGE на тесте: 75.70%
```

Учитывая неблагоприятное разбиение на обучающий и тестовый наборы данных, мы получаем приличную правильность на тесте, равную 74.70 %. Однако GraphSAGE демонстрирует более низкую правильность по сравнению с GCN или GAT на наборе данных `PubMed`. Так почему же мы должны использовать эту модель? Ответ становится очевидным, когда вы обучаете три модели: GraphSAGE работает крайне быстро. На обычном GPU модель обучается в 4 раза быстрее, чем GCN, и в 88 раз быстрее, чем GAT. Даже если скорость обучения не является проблемой, GraphSAGE дает лучшее качество для больших графов, нежели для небольших сетей. Чтобы завершить этот глубокий анализ архитектуры GraphSAGE, мы должны обсудить еще одну ее особенность – ее индуктивные возможности.

Индуктивное обучение на белок-белковых взаимодействиях

В GNN мы различаем два типа обучения – **трансдуктивное** (transductive) и **индуктивное** (inductive). Их можно резюмировать следующим образом:

- при индуктивном обучении графовая нейронная сеть во время обучения видит только данные обучающего набора. Это типичное машинное обучение с учителем. В этой ситуации метки используются для настройки параметров GNN;
- при трансдуктивном обучении графовая нейронная сеть во время обучения видит данные обучающего и тестового наборов (видит узлы из тестового набора, но при этом не имеет доступ к их меткам). В этой ситуации метки используются для распространения информации.

В трансдуктивном обучении обучающие, проверочные и тестовые наборы относятся к одному и тому же графу. Объединив все наборы, сможем получить весь граф. В индуктивном обучении обучающие, проверочные и тестовые наборы представлены разными графами.

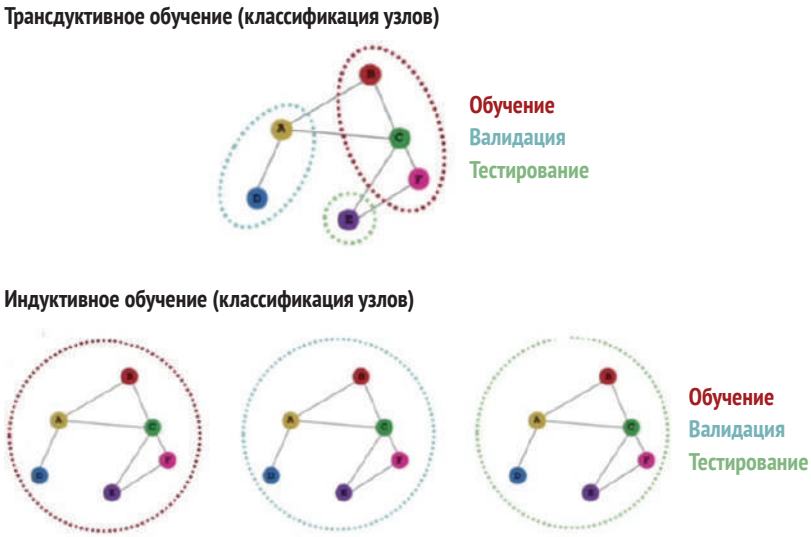


Рис. 8.6 ❖ Визуализация трансдуктивного обучения и индуктивного обучения для задачи классификации узлов

Трансдуктивное обучение должно быть вам знакомо, поскольку это единственное обучение, которое мы рассматривали до сих пор. Действительно в предыдущем примере можно увидеть, что GraphSAGE делает прогнозы, используя весь граф во время обучения (`self(batch.x, batch.edge_index)`). Затем мы маскируем часть этих прогнозов, чтобы вычислить потери и обучаем модель только с использованием обучающих данных (`criterion(out[batch.train_mask], batch.y[batch.train_mask])`).

Трансдуктивное обучение может генерировать эмбеддинги только для одного зафиксированного графа. Оно не может обобщать на узлы или графы, которых не было в обучении. Однако благодаря семплированию соседей GraphSAGE предназначен для прогнозирования на локальном уровне (делает прогнозы для каждой вершины графа, основываясь на ее локальном окруже-

нии, т. е. на ее соседях) с использованием сокращенных графов вычислений. Это считается индуктивной стратегией, поскольку ее можно применять к любому графу вычислений с теми же самими признаками.

Давайте применим индуктивное обучение к новому набору данных `PPI` – сети белок-белкового взаимодействия (protein-protein interaction – PPI), описанной Agrawal et al. [5]. Этот набор данных представляет собой набор из 24 графов, где узлы (21 557) – это человеческие белки, а ребра (342 353) – физические взаимодействия между белками в человеческой клетке. На рис. 8.7 показана визуализация набора `PPI`, созданная с помощью Gephi.

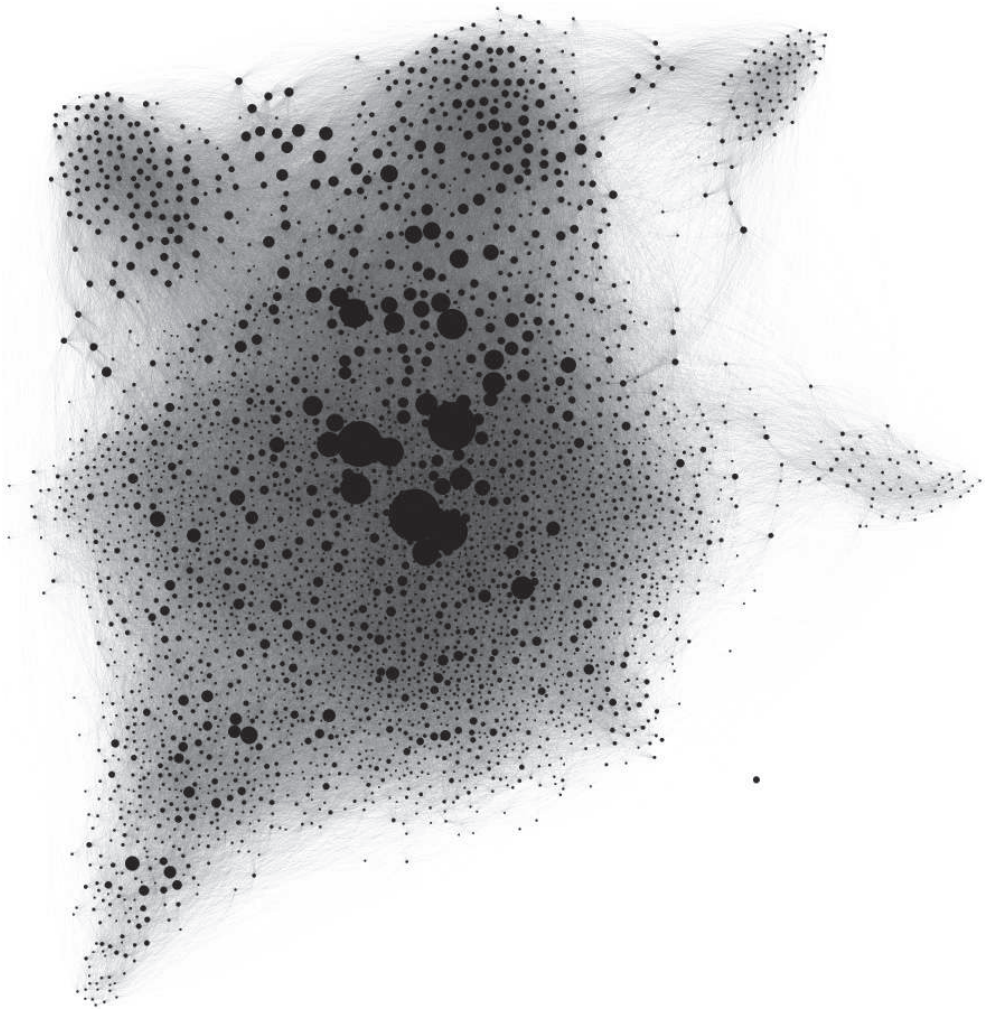


Рис. 8.7 ❖ Визуализация сети белок-белковых взаимодействий

Цель набора данных – выполнить многометочную классификацию (121 метка). Таким образом, узел может быть отнесен к нескольким классам

одновременно. Это отличается от многоклассовой классификации, где каждый узел имеет только один класс.

Давайте реализуем новую модель GraphSAGE с использованием PyG.

1. Импортируем необходимые библиотеки, классы, функции и загружаем набор данных PPI, при этом, если доступен GPU, используем его, в противном случае используем GPU:

```
set_seed()

from sklearn.metrics import f1_score

from torch_geometric.datasets import PPI
from torch_geometric.data import Batch
from torch_geometric.loader import DataLoader
from torch_geometric.nn import GraphSAGE

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

2. Формируем обучающий, валидационный и тестовый наборы. Обратите внимание, обучающий набор содержит 20 графов, тогда как валидационный и тестовый наборы содержат по 2 графа:

```
# формируем обучающий, валидационный и тестовый наборы
train_dataset = PPI(root=".", split='train')
val_dataset = PPI(root=".", split='val')
test_dataset = PPI(root=".", split='test')
```

3. Объединяем графы обучающего набора:

```
# объединяем графы обучающего набора
train_data = Batch.from_data_list(train_dataset)
```

4. Теперь создаем батчи с семплированием соседей:

```
# создаем батчи с семплированием соседей
train_loader = NeighborLoader(
    train_data, batch_size=2048,
    shuffle=True, num_neighbors=[20, 10],
    num_workers=2, persistent_workers=True
)
```

5. Теперь создаем загрузчики данных для валидации и тестирования (по два графа в батче):

```
# создаем загрузчики данных для валидации и тестирования
# (два графа в батче)
val_loader = DataLoader(val_dataset, batch_size=2)
test_loader = DataLoader(test_dataset, batch_size=2)
```

6. Вместо того чтобы реализовывать модель GraphSAGE самостоятельно, мы можем непосредственно взять ее реализацию из PyTorch Geometric, доступную в модуле torch_geometric.nn. Возьмем два слоя и установим

количество скрытых измерений равным 512. Кроме того, нам нужно перенести модель на выбранное устройство, используя `.to(device)`:

```
# задаем модель GraphSAGE
model = GraphSAGE(
    in_channels=train_dataset.num_features,
    hidden_channels=512,
    num_layers=2,
    out_channels=train_dataset.num_classes
).to(device)
```

7. Функция `fit()` аналогична той, которую мы использовали в предыдущем разделе, за исключением двух моментов. Во-первых, мы хотим, когда это возможно, использовать GPU. Во-вторых, у нас приходится по 2 графа на батч, поэтому мы умножаем значение функции потерь на два (`data.num_graphs`):

```
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)

def fit(loader):
    model.train()

    total_loss = 0
    for data in loader:
        data = data.to(device)
        optimizer.zero_grad()
        out = model(data.x, data.edge_index)
        loss = criterion(out, data.y)
        total_loss += loss.item() * data.num_graphs
        loss.backward()
        optimizer.step()
    return total_loss / len(loader.data)
```

8. В функции `test()` мы воспользовались тем фактом, что в `val_loader` и `test_loader` используется `batch_size=2`. Это означает, что два графа находятся в одном и том же батче, нам не нужно итерировать по загрузчику, как мы это делали во время обучения. Вместо правильности давайте воспользуемся другой метрикой – F1-мерой. Оно соответствует среднему гармоническому среднему точности и полноты. Однако наши прогнозы представляют собой 121-мерные векторы действительных чисел. Нам нужно преобразовать их в бинарные векторы, используя `out > 0` для сравнения их с `data.y`:

```
@torch.no_grad()
def test(loader):
    model.eval()

    data = next(iter(loader))
    out = model(data.x.to(device), data.edge_index.to(device))
    preds = (out > 0).float().cpu()
```

```
    y, pred = data.y.numpy(), preds.numpy()
    return f1_score(y, pred, average='micro') if pred.sum() > 0 else 0
```

9. Давайте обучим нашу модель в течение 300 эпох, выводя значение F1-меры во время обучения:

```
for epoch in range(301):
    loss = fit(train_loader)
    val_f1 = test(val_loader)
    if epoch % 50 == 0:
        print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: '
              f'{loss:.3f} | F1 на валид. выборке: {val_f1:.4f}')
```

```
Эпоха   0:
| Функция потерь на обуч. выборке: 12.720 | F1 на валид. выборке: 0.4883
Эпоха  50:
| Функция потерь на обуч. выборке: 8.748 | F1 на валид. выборке: 0.7990
Эпоха 100:
| Функция потерь на обуч. выборке: 8.607 | F1 на валид. выборке: 0.8119
Эпоха 150:
| Функция потерь на обуч. выборке: 8.532 | F1 на валид. выборке: 0.8199
Эпоха 200:
| Функция потерь на обуч. выборке: 8.496 | F1 на валид. выборке: 0.8246
Эпоха 250:
| Функция потерь на обуч. выборке: 8.494 | F1 на валид. выборке: 0.8265
Эпоха 300:
| Функция потерь на обуч. выборке: 8.459 | F1 на валид. выборке: 0.8186
```

10. Наконец, вычислим значение F1-меры для тестового набора:

```
print(f'F1 на тесте: {test(test_loader):.4f}')
```

```
F1 на тесте: 0.8420
```

Мы получили хорошее значение F1-меры 0.842 в режиме индуктивного обучения. Это значение резко меняется при увеличении или уменьшении размера скрытых каналов. Вы можете попробовать сами, используя другие значения, например 128 или 1024 вместо 512.

Если вы внимательно посмотрите на программный код, то увидите, что маски не используются. Действительно индуктивное обучение обусловлено набором данных `PPI`: данные для обучения, валидации и тестирования находятся в разных графах и загрузчиках. Естественно, мы могли бы объединить их с помощью `Batch.from_data_list()` и вернуться к трансдуктивному обучению.

Кроме того, мы могли бы обучить GraphSAGE без меток, используя обучение без учителя. Это особенно полезно, когда меток мало или они связаны с другими задачами. Однако для этого требуется новая функция потерь, которая будет поощрять схожесть представлений для соседних узлов, гарантируя при этом, что удаленные друг от друга узлы будут иметь сильно отличающиеся представления:

$$J_G(h_i) = -\log(\sigma(h_i^T h_j)) - Q \cdot E_{j_n \sim P_n(j)} \log(\sigma(-h_i^T h_{j_n})).$$

Здесь j – сосед узла u в случайном блуждании, σ – сигмоидная функция, $P_n(j)$ – распределение отрицательного семплирования для j и Q – это количество отрицательных примеров.

Положительные примеры – это пары узлов, для которых в графе существует связь. Такие примеры представляют собой «правильные», или «положительные», связи. Отрицательные примеры – это пары узлов, для которых связи в графе отсутствуют. Эти примеры формируются путем случайного семплирования узлов и создания их пар, предполагая, что между ними нет связи. Отрицательные примеры представляют собой «неправильные», или «отрицательные», связи. В процессе обучения GraphSAGE модель может использовать как положительные, так и отрицательные примеры для оптимизации параметров. Это помогает модели лучше понимать, как выявлять существующие связи и различать их от случайных соединений узлов. Такой подход в обучении с использованием отрицательных примеров помогает улучшить обобщающую способность модели и ее способность обрабатывать новые данные, не участвовавшие в обучении.

Наконец, PinSAGE и вариант GraphSAGE от Uber Eats представляют собой рекомендательные системы. Они используют обучение без учителя и другую функцию потерь в силу особенностей задачи. Их целью является ранжирование наиболее релевантных сущностей (продуктов, ресторанов, пинов и т. д.) для каждого пользователя, что представляет собой совершенно другую задачу. Для решения этой задачи они используют функцию потерь, ранжирующую на основе максимального зазора, которая рассматривает пары эмбеддингов.

Если вам нужно масштабировать графовые нейросети, можно рассмотреть другие решения. Вот краткое описание двух стандартных техник:

- Cluster-GCN [6]: предлагает другой ответ на вопрос о том, как создавать мини-батчи. Вместо семплирования соседей граф делится на изолированные сообщества. Эти сообщества затем обрабатываются как независимые графы, что может негативно сказаться на качестве полученных эмбеддингов;
- упрощение архитектуры графовых нейронных сетей: может снизить время обучения и инференса. На практике упрощение заключается в отказе от нелинейных функций активации. Линейные слои можно свести к одной операции матричного умножения с использованием линейной алгебры. Естественно, эти упрощенные версии не так точны на небольших наборах данных, но эффективны для больших графов, таких как Twitter [7].

Как видите, GraphSAGE – это гибкий фреймворк, который можно править и настраивать под свои цели. Даже если вы не будете его в дальнейшем повторно использовать, он вводит ключевые понятия, которые существенно влияют на архитектуры графовых нейронных сетей в целом.

Выводы

В этой главе мы познакомились с фреймворком GraphSAGE и его двумя компонентами – алгоритмом семплирования соседей и тремя операторами агрегирования. Главным образом семплирование соседей позволяет GraphSAGE обрабатывать большие графы за короткое время. Оно также делает возможным индуктивное обучение, которое позволяет обобщать прогнозы на новые узлы и графы. Мы осуществили трансдуктивное обучение на данных `PubMed` и индуктивное обучение для выполнения новой задачи на наборе данных `PPI` – многометочной классификации. Хотя GraphSAGE не так точен, как GCN или GAT, он является популярным и эффективным фреймворком для обработки огромных объемов данных.

В главе 9 «Определение выразительности для классификации графов» мы постараемся определить, что делает графовую нейронную сеть мощной с точки зрения представлений. Мы представим известный графовый алгоритм, называемый тестом изоморфизма Вайсфейлера–Лемана. Он будет использоваться в качестве эталона для оценки теоретической эффективности различных архитектур графовой нейронной сети, включая графовую сеть изоморфизма. Мы применим эту графовую нейронную сеть для выполнения новой актуальной задачи – классификации графов.

Дополнительное чтение

[1] W. L. Hamilton, R. Ying, and J. Leskovec. *Inductive Representation Learning on Large Graphs*. arXiv, 2017. DOI: 10.48550/ARXIV.1706.02216.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. *Graph Convolutional Neural Networks for Web-Scale Recommender Systems*. Jul. 2018. DOI: 10.1145/3219819.3219890.

[3] Ankit Jain. *Food Discovery with Uber Eats: Using Graph Learning to Power Recommendations*: https://www.uber.com/en-US/blog/uber-eats-graph-learning/.

[4] Galileo Mark Namata, Ben London, Lise Getoor, and Bert Huang. *Query-Driven Active Surveying for Collective Classification*. International Workshop on Mining and Learning with Graphs. 2012.

[5] M. Agrawal, M. Zitnik, and J. Leskovec. *Large-scale analysis of disease pathways in the human interactome*. Nov. 2017. DOI: 10.1142/9789813235533_0011.

[6] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh. *Cluster-GCN*. Jul. 2019. DOI: 10.1145/3292500.3330925.

[7] F. Frasca, E. Rossi, D. Eynard, B. Chamberlain, M. Bronstein, and F. Monti. *SIGN: Scalable Inception Graph Neural Networks*. arXiv, 2020. DOI: 10.48550/ARXIV.2004.11198.

Глава 9

Определение выразительности для классификации графов

В предыдущей главе мы попробовали найти компромисс между качеством модели и масштабируемостью. Мы увидели, что это полезно в таких сферах применения, как рекомендательные системы. Однако возникает несколько вопросов, касающихся «точности» GNN. Чем обусловлена эта «точность»? Можем ли мы использовать эти знания для разработки GNN лучшего качества?

В этой главе мы объясним, что делает GNN мощной моделью, прибегнув к **тесту Вейсфейлера–Лемана** (Weisfeiler-Leman – WL). Этот тест познакомит нас с важным понятием теории GNN – **выразительностью** (expressiveness). Мы воспользуемся им, чтобы сравнить разные слои GNN и посмотреть, какой из них обладает наибольшей выразительностью. Затем полученный результат будет использован для разработки GNN, более мощной, чем GCN, GAT и GraphSAGE.

Наконец, мы реализуем новую модель GNN с помощью PyTorch Geometric для выполнения новой задачи – классификации графов. Мы реализуем ее для набора данных `PROTEINS`, состоящего из 1113 графов, представляющих белки. Мы сравним различные методы классификации графов и проанализируем наши результаты.

К концу этой главы вы поймете, что делает GNN выразительной и как измерить выразительность. Вы сможете реализовать новую архитектуру GNN на основе теста WL и выполнить классификацию графов с использованием различных методов.

В этой главе мы рассмотрим следующие основные темы:

- «Определение выразительности»,
- «Знакомство с графовой сетью изоморфизма»,
- «Классификация графов с помощью графовой сети изоморфизма».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter09.

Определение выразительности

Нейронные сети применяются для аппроксимации функций. Обоснованием этому является **универсальная теорема аппроксимации** (universal approximation theorem), которая утверждает, что однослойная прямая нейронная сеть прямой связи способна аппроксимировать любую гладкую функцию. Но как обстоит дело с универсальной аппроксимацией функций, когда речь заходит о графах? Эта задача более сложная, и здесь требуется способность различать структуры графов.

Когда мы работаем с графовыми нейронными сетями, наша цель заключается в том, чтобы получить наилучшие эмбеддинги узлов. Это подразумевает, что отличающиеся друг от друга узлы должны давать отличающиеся друг от друга эмбеддинги, а похожие друг на друга узлы – похожие друг на друга эмбеддинги. Однако как мы можем определить сходство двух узлов? Эмбеддинги вычисляются с использованием информации о характеристиках узлов и их связей. Следовательно, чтобы отличать узлы друг от друга, нам необходимо сравнить характеристики узлов и их соседей.

В терминологии теории графов это известно как проблема **изоморфизма** (isomorphism) графов. Два графа считаются изоморфными («одинаковыми»), если у них одинаковые связи, и единственное различие заключается в перестановке их узлов (см. рис. 9.1). В 1968 году Вайсфейлер и Леман [1] предложили эффективный алгоритм для решения этой проблемы, который сегодня известен как тест Вайсфейлера–Лемана (WL).

Тест Вайсфейлера–Лемана нацелен на создание **канонической формы** (canonical form) графа[1]. Затем мы можем сравнить канонические формы двух

[1] Каноническая форма графа – это представление графа, выбранное из множества его эквивалентных представлений таким образом, что оно служит своего рода уникальным «идентификатором», или «отпечатком» графа. Эта форма создается с учетом определенных критериев, которые обеспечивают уникальность и сравнимость графов.

графов, чтобы определить, изоморфны они или нет. Тем не менее этот тест не является идеальным, и неизоморфные графы могут иметь одну и ту же каноническую форму. Это может показаться неожиданным, но речь идет о сложной проблеме, которая до сих пор не решена полностью, например неизвестна сложность алгоритма Вайсфейлера–Лемана.

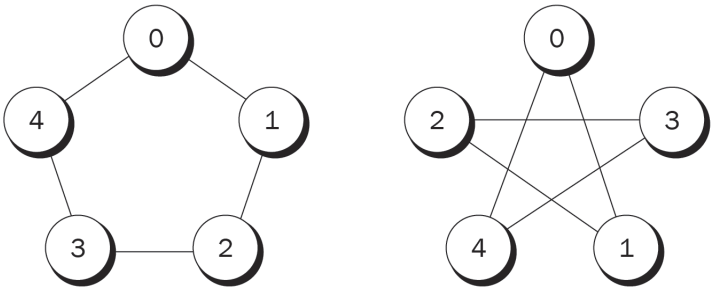


Рис. 9.1 ❖ Пример двух изоморфных графов

Тест Вайсфейлера–Лемана работает следующим образом.

1. На начальном этапе каждый узел в графе получает одинаковый цвет.
2. Каждый узел агрегирует информацию о своем собственном цвете и цвете своих соседей.
3. Результат передается хеш-функции, которая создает новый цвет.
4. Каждый узел агрегирует информацию о своем новом цвете и новых цветах своих соседей.
5. Результат передается хеш-функции, которая создает новый цвет.
6. Эти шаги повторяются до тех пор, пока цвета узлов не перестанут меняться.

На рис. 9.2 кратко представлен алгоритм WL.

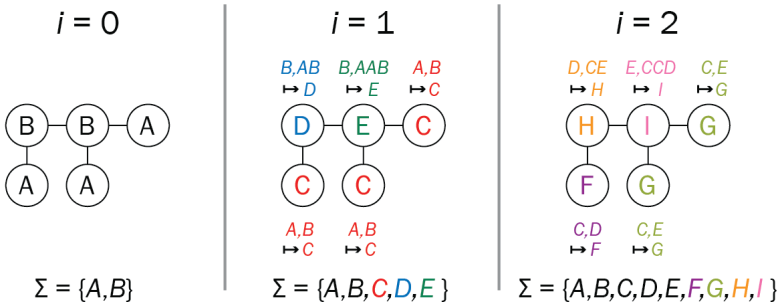


Рис. 9.2 ❖ Применение алгоритма Вайсфейлера–Лемана для получения канонической формы графа

Полученные цвета дают нам каноническую форму графа. Если два графа получили разные цвета, они не изоморфны. Однако обратное утверждение

не всегда верно: два графа, получивших одни и те же цвета, могут быть изоморфными, но могут и не быть таковыми.

Шаги, которые мы описали, должны быть вам знакомы, они на удивление близки к операциям, которые выполняют графовые нейронные сети. Цвета – это вид эмбеддингов, а хеш-функция – это агрегатор. Но это не просто агрегатор, хеш-функция здесь особенно подходит для этой задачи. Была бы она такой же эффективной, если бы мы заменили ее другой функцией, например агрегатором на основе среднего или максимального значения (как рассказывалось в главе 8)?

Давайте посмотрим результат для каждого оператора.

Допустим, используем агрегатор на основе среднего значения. Если у нас 1 синий узел и 1 красный узел или 10 синих узлов и 10 красных узлов, то получится один и тот же эмбеддинг (наполовину синего цвета, наполовину красного).

При использовании агрегатора на основе максимального значения половина узлов будет игнорироваться, эмбеддинг будет учитывать либо только синий цвет, либо только красный.

Однако при использовании агрегатора на основе суммы каждый узел вносит свой вклад в итоговый эмбеддинг: структура с 1 красным узлом и 1 синим узлом отличается от структуры с 10 синими узлами и 10 красными узлами.

Действительно, агрегатор сумм может различать больше структур графа, чем два других. Если следовать этой логике, это может означать только одно – агрегаторы, которые мы использовали до сих пор, неоптимальны, поскольку они строго менее выразительны, чем агрегатор на основе сумм. Можем ли мы использовать эти знания для создания более оптимальных графовых нейронных сетей? В следующем разделе мы представим **графовую сеть изоморфизма** (Graph Isomorphism Network – GIN), основанную на этой идее.

Знакомство с графовой сетью изоморфизма

В предыдущем разделе мы убедились в том, что графовые нейронные сети, представленные в предыдущих главах, были менее выразительными, чем тест Вайсфейлера–Лемана. Это проблема, поскольку способность различать большее количество графовых структур, по-видимому, связана с качеством получаемых эмбеддингов. В этом разделе мы перейдем от теоретической основы к новой архитектуре графовых нейронных сетей – графовой сети изоморфизма.

В 2018 году Сюй совместно с коллегами в статье «How Powerful are Graph Neural Networks?» [2] представил архитектуру GIN, спроектированную так, чтобы быть такой же выразительной, как и тест Вайсфейлера–Лемана. Авторы обобщили теорию по процедуре агрегации, разбив ее на две функции:

- **агрегировать** (aggregate): функция f отбирает соседние узлы, рассматриваемые графовой нейронной сетью;
- **комбинировать** (combine): функция ϕ комбинирует эмбеддинги отобранных узлов для создания нового эмбеддинга целевого узла.

Эмбеддинг узла i можно записать следующим образом:

$$h_i' = \phi(h_i, f(\{h_j : j \in \mathcal{N}_i\})).$$

В случае графовой сверточной сети функция f агрегирует информацию о каждом соседе узла i, а функция ϕ применяет специальный агрегатор на основе среднего значения. В случае GraphSAGE семлирование соседей – это функция, и мы видели три варианта ϕ – агрегаторы на основе среднего значения, LSTM и максимального значения.

Итак, что же это за функции в графовой нейронной сети? Сюй и коллеги утверждают, что эти функции должны быть **инъективными** (injective). Как показано на рис. 9.3, инъективные функции отображают разные входные данные в разные выходные данные. Это именно то, что нам нужно, чтобы отличать графовые структуры друг от друга. Если бы функции не были инъективными, мы бы получили один и тот же результат для разных входящих элементов. В этом случае наши эмбеддинги будут менее ценными, поскольку будут содержать меньше информации.

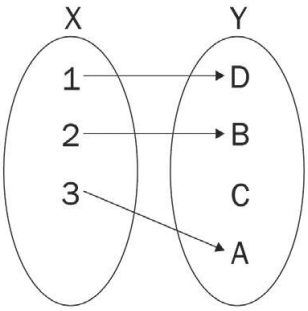


Рис. 9.3 ❖ Отображающая диаграмма для инъективной функции

Авторы GIN используют хитрый трюк, аппроксимируя эти две функции. В GAT-слое мы выучиваем веса самовнимания. В этом примере мы можем обучить обе функции, используя многослойный перцептрон (MLP) благодаря универсальной теореме аппроксимации:

$$h_i' = \text{MLP}\left((1+\varepsilon)\cdot h_i + \sum_{j\in\mathcal{N}_i} h_j\right).$$

Здесь ε – выучиваемый параметр или зафиксированное скалярное значение, представляющее собой важность эмбеддинга целевого узла по срав-

нению с эмбеддингами его соседей. Авторы также подчеркивают, что MLP должен иметь более одного слоя, чтобы различать конкретные структуры графов.

Теперь мы получаем графовую сеть изоморфизма, обладающую такой же выразительностью, как и тест Вайсфейлера–Лемана. Можем ли мы добиться еще лучшего результата? Ответ: да. Тест Вайсфейлера–Лемана можно обобщить на иерархию тестов более высокого уровня, известную как **k-WL**. Вместо рассмотрения отдельных узлов k-WL тесты рассматривают кортежи узлов (k – это длина кортежа). Это означает, что они нелокальны, поскольку могут просматривать удаленные узлы. Именно поэтому (k + 1)-WL-тесты могут различать больше графовых структур, чем k-WL-тесты для $k \geq 2$.

Было предложено несколько архитектур, основанных на k-WL-тестах типа **k-GNN**, предложенного Моррисом совместно с коллегами [3]. Хотя эти архитектуры помогают нам лучше понять, как работают графовые нейронные сети, на практике они, как правило, уступают менее выразительным моделям, таким как графовые нейронные сети или графовые нейронные сети с вниманием [4]. Но, как мы увидим в следующем разделе в контексте классификации графов, надежда еще не потеряна.

Классификация графов с помощью графовой сети изоморфизма

Мы могли бы напрямую реализовать модель GIN для классификации узлов, но эта архитектура более интересна для классификации графов. В этом разделе мы увидим, как преобразовать эмбеддинги узлов в эмбеддинги графов, используя методы **глобального пулинга** (global pooling). Затем применим эти методы к набору данных PROTEINS и сравним наши результаты, применив модели GIN и GCN.

Классификация графов

Классификация графов основывается на эмбеддингах узлов, которые создает GNN. Эту операцию часто называют **глобальным пулингом** или **считыванием на уровне графа** (graph-level readout). Есть три простых способа реализовать его.

- **Глобальный пулинг на основе среднего** (mean global pooling)**.** Эмбеддинг графа h_G получается путем усреднения эмбеддингов каждого узла графа:

$$h_G = \frac{1}{N}\sum_{i=0}^{N} h_i.$$

- **Глобальный пулинг на основе максимального значения** (max global pooling). Эмбеддинг графа получается путем выбора наибольшего значения для каждой размерности узла:

$$h_G = \max_{i=0}^{N}(h_i).$$

- **Глобальный пулинг на основе суммы**, или **глобальный пулинг путем суммирования** (sum global pooling). Эмбеддинг графа получается путем суммирования эмбеддингов каждого узла графа:

$$h_G = \sum_{i=0}^{N} h_i.$$

Согласно тому, что мы видели в первом разделе, глобальный пулинг на основе суммы является строго более выразительным, чем два других метода. Авторы графовой сети изоморфизма (GIN) также отмечают, что для учета всей структурной информации необходимо рассмотреть эмбеддинги, созданные каждым слоем графовой нейронной сети (GNN). В итоге мы конкатенируем суммы эмбеддингов узлов, созданных каждым из k слоев нашей GNN:

$$h_G = \sum_{i=0}^{N} h_i^0 \,||\, \ldots \,||\, \sum_{i=0}^{N} h_i^k.$$

Это решение элегантно сочетает в себе выразительную мощь оператора суммирования и использование памяти каждого слоя за счет конкатенации.

Реализация графовой сети изоморфизма (GIN)

Теперь мы реализуем модель графовой сети изоморфизма (GIN) с ранее упомянутой функцией считывания на уровне графа для набора данных `PROTEINS`.

Этот набор данных содержит 1113 графов, представляющих белки, в которых каждый узел представляет собой аминокислоту. Ребро соединяет два узла, когда расстояние между ними меньше 0.6 нм. Задача заключается в том, чтобы классифицировать каждый белок как **фермент** (enzyme). Ферменты – это специфические белки, которые действуют как катализаторы, ускоряя химические реакции в клетке. Например, ферменты, называемые липазами, помогают переваривать пищу. На рис. 9.4 показан трехмерный граф белка.

Давайте реализуем модель GIN на этом наборе данных.

1. Зададим некоторые настройки и напишем собственную функцию для обеспечения воспроизводимости результатов:

```
import torch
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

```
def set_seed():
    """
    Задает стартовое значение генератора псевдослучайных
    чисел для воспроизводимости.
    """
    torch.manual_seed(-1)
    torch.cuda.manual_seed(0)
    torch.cuda.manual_seed_all(0)
```

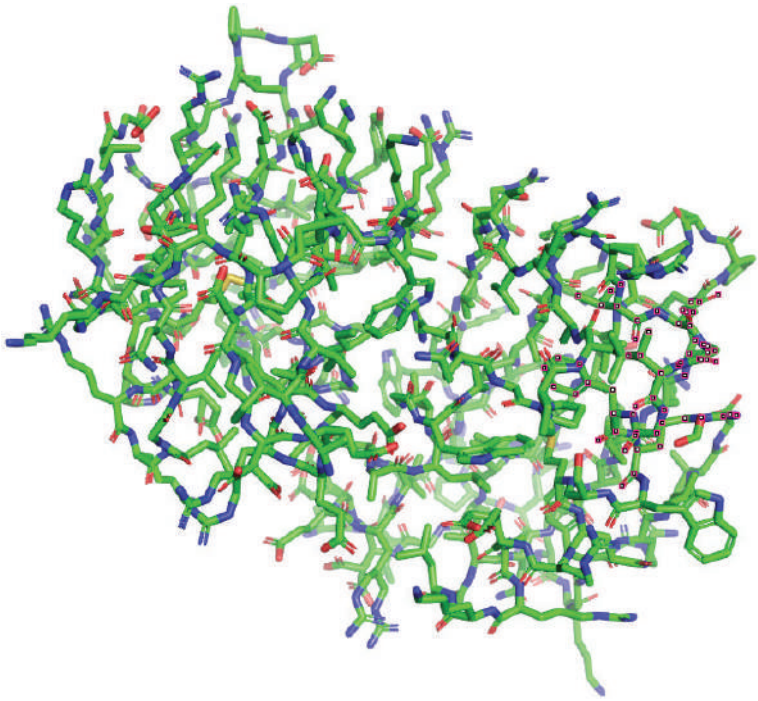


Рис. 9.4 ❖ Трехмерный граф белка

2. Импортируем набор данных `PROTEINS` с помощью класса `TUDataset` библиотеки PyTorch Geometric и выведем информацию о нем:

```
from torch_geometric.datasets import TUDataset

dataset = TUDataset(root='.', name='PROTEINS').shuffle()

# печатаем информацию о наборе данных
print(f'Набор данных: {dataset}')
print('----------------------')
print(f'Количество графов: {len(dataset)}')
print(f'Количество узлов: {dataset[0].x.shape[0]}')
print(f'Количество признаков: {dataset.num_features}')
print(f'Количество классов: {dataset.num_classes}')
```

```
Набор данных: PROTEINS(1113)
----------------------
Количество графов: 1113
Количество узлов: 27
Количество признаков: 3
Количество классов: 2
```

3. Разбиваем данные (графы) на обучающий, проверочный и тестовый наборы, используя пропорции 80/10/10 соответственно:

```
from torch_geometric.loader import DataLoader

# создаем обучающий, валидационный и тестовый наборы
train_dataset = dataset[:int(len(dataset)*0.8)]
val_dataset   = dataset[int(len(dataset)*0.8):int(len(dataset)*0.9)]
test_dataset  = dataset[int(len(dataset)*0.9):]

print(f'Обучающий набор     = {len(train_dataset)} графов')
print(f'Валидационный набор = {len(val_dataset)} графов')
print(f'Тестовый набор      = {len(test_dataset)} графов')
```

```
Обучающий набор     = 890 графов
Валидационный набор = 111 графов
Тестовый набор      = 112 графов
```

4. Мы преобразуем эти наборы в мини-батчи с помощью объекта `DataLoader`, задав размер батча равным 64. Это означает, что каждый батч будет содержать до 64 графов:

```
# создаем мини-батчи
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader   = DataLoader(val_dataset, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=64, shuffle=True)
```

5. Мы можем убедиться в этом, распечатав информацию о каждом батче следующим образом:

```
print('\nЗагрузчик обучающих данных:')
for i, batch in enumerate(train_loader):
    print(f' - Батч {i}: {batch}')

print('\nЗагрузчик валидационных данных:')
for i, batch in enumerate(val_loader):
    print(f' - Батч {i}: {batch}')

print('\nЗагрузчик тестовых данных:')
for i, batch in enumerate(test_loader):
    print(f' - Батч {i}: {batch}')
```

```
Загрузчик обучающих данных:
 - Батч 0: DataBatch(edge_index=[2, 10078], x=[2641, 3], y=[64], batch=[2641],
ptr=[65])
 - Батч 1: DataBatch(edge_index=[2, 8412], x=[2185, 3], y=[64], batch=[2185],
ptr=[65])
```

```
 - Батч 2: DataBatch(edge_index=[2, 8240], x=[2117, 3], y=[64], batch=[2117],
ptr=[65])
 - Батч 3: DataBatch(edge_index=[2, 11730], x=[3106, 3], y=[64], batch=[3106],
ptr=[65])
 - Батч 4: DataBatch(edge_index=[2, 7932], x=[2110, 3], y=[64], batch=[2110],
ptr=[65])
 - Батч 5: DataBatch(edge_index=[2, 8716], x=[2346, 3], y=[64], batch=[2346],
ptr=[65])
 - Батч 6: DataBatch(edge_index=[2, 8820], x=[2412, 3], y=[64], batch=[2412],
ptr=[65])
 - Батч 7: DataBatch(edge_index=[2, 13188], x=[3549, 3], y=[64], batch=[3549],
ptr=[65])
 - Батч 8: DataBatch(edge_index=[2, 8268], x=[2209, 3], y=[64], batch=[2209],
ptr=[65])
 - Батч 9: DataBatch(edge_index=[2, 8190], x=[2274, 3], y=[64], batch=[2274],
ptr=[65])
 - Батч 10: DataBatch(edge_index=[2, 8484], x=[2287, 3], y=[64], batch=[2287],
ptr=[65])
 - Батч 11: DataBatch(edge_index=[2, 9944], x=[2664, 3], y=[64], batch=[2664],
ptr=[65])
 - Батч 12: DataBatch(edge_index=[2, 11162], x=[3133, 3], y=[64], batch=[3133],
ptr=[65])
 - Батч 13: DataBatch(edge_index=[2, 7244], x=[1977, 3], y=[58], batch=[1977],
ptr=[59])

Загрузчик валидационных данных:
 - Батч 0: DataBatch(edge_index=[2, 10138], x=[2810, 3], y=[64], batch=[2810],
ptr=[65])
 - Батч 1: DataBatch(edge_index=[2, 5170], x=[1387, 3], y=[47], batch=[1387],
ptr=[48])

Загрузчик тестовых данных:
 - Батч 0: DataBatch(edge_index=[2, 8502], x=[2283, 3], y=[64], batch=[2283],
ptr=[65])
 - Батч 1: DataBatch(edge_index=[2, 7870], x=[1981, 3], y=[48], batch=[1981],
ptr=[49])
```

6. Приступим к программным реализациям GCN и GIN. Что касается GIN, то сперва нужно определиться со структурой нашего GIN-слоя. Нам нужен многослойный перцептрон (MLP) как минимум с двумя слоями. Следуя рекомендациям авторов, мы также можем внедрить батч-нормализацию для стандартизации входов каждого скрытого слоя, что стабилизирует и ускоряет обучение. В итоге наш GIN-слой будет иметь следующую структуру:

$$\text{Linear} \rightarrow \text{BatchNorm} \rightarrow \text{ReLu} \rightarrow \text{Linear} \rightarrow \text{ReLu}.$$

В ходе реализации GIN не следует забывать, что мы хотим выполнить классификацию графов. Для этого необходима сумма эмбеддингов каждого узла графа для каждого слоя. Другими словами, нам придется хранить для каждого из трех слоев один вектор размером `dim_h`. По-

этому мы добавляем линейный слой размером `3*dim_h` перед итоговым линейным слоем для бинарной классификации (`data.num_classes = 2`). Каждый слой создает разные тензоры эмбеддингов – `h1`, `h2` и `h3`. Мы суммируем их, используя функцию `global_add_pool()`, а затем конкатенируем с помощью `torch.cat()`. Это дает нам входные данные для нашего классификатора, который работает как обычная нейронная сеть с дропаутом.

```
import torch.nn.functional as F
from torch.nn import Linear, Sequential, BatchNorm1d, ReLU, Dropout
from torch_geometric.nn import GCNConv, GINConv
from torch_geometric.nn import global_mean_pool, global_add_pool


class GCN(torch.nn.Module):
    """GCN"""
    def __init__(self, dim_h):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(dataset.num_node_features, dim_h)
        self.conv2 = GCNConv(dim_h, dim_h)
        self.conv3 = GCNConv(dim_h, dim_h)
        self.lin = Linear(dim_h, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        # эмбеддинги узлов
        h = self.conv1(x, edge_index)
        h = h.relu()
        h = self.conv2(h, edge_index)
        h = h.relu()
        h = self.conv3(h, edge_index)

        # считывание (readout) на уровне графа
        hG = global_mean_pool(h, batch)

        # классификатор
        h = F.dropout(hG, p=0.5, training=self.training)
        h = self.lin(h)

        return F.log_softmax(h, dim=1)

class GIN(torch.nn.Module):
    """GIN"""
    def __init__(self, dim_h):
        super(GIN, self).__init__()
        self.conv1 = GINConv(
            Sequential(Linear(dataset.num_node_features, dim_h),
                       BatchNorm1d(dim_h), ReLU(),
                       Linear(dim_h, dim_h), ReLU()))
        self.conv2 = GINConv(
            Sequential(Linear(dim_h, dim_h), BatchNorm1d(dim_h), ReLU(),
                       Linear(dim_h, dim_h), ReLU()))
```

```
        self.conv3 = GINConv(
            Sequential(Linear(dim_h, dim_h), BatchNorm1d(dim_h), ReLU(),
                       Linear(dim_h, dim_h), ReLU()))
        self.lin1 = Linear(dim_h*3, dim_h*3)
        self.lin2 = Linear(dim_h*3, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        # эмбеддинги узлов
        h1 = self.conv1(x, edge_index)
        h2 = self.conv2(h1, edge_index)
        h3 = self.conv3(h2, edge_index)

        # считывание (readout) на уровне графа
        h1 = global_add_pool(h1, batch)
        h2 = global_add_pool(h2, batch)
        h3 = global_add_pool(h3, batch)

        # конкатенируем эмбеддинги графов
        h = torch.cat((h1, h2, h3), dim=1)

        # классификатор
        h = self.lin1(h)
        h = h.relu()
        h = F.dropout(h, p=0.5, training=self.training)
        h = self.lin2(h)

        return F.log_softmax(h, dim=1)
```

Примечание

Кроме того, PyTorch Geometric предлагает GINE-слой, модифицированную версию GIN-слоя. Он был представлен в 2019 году Xy совместно с коллегами в статье «Strategies for Pre-training Graph Neural Networks» [8]. Его основным улучшением, по сравнению с предыдущей GIN-версией, является возможность учитывать в процессе агрегации признаки ребер. Набор данных PROTEINS не имеет признаков ребер, поэтому мы реализуем классическую модель GIN.

7. Теперь можем реализовать обычный цикл обучения с использованием мини-батчей, задав 100 эпох:

```
set_seed()

def train(model, loader):
    criterion = torch.nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    epochs = 100

    model.train()
    for epoch in range(epochs+1):
        total_loss = 0
        acc = 0
```

```
        val_loss = 0
        val_acc = 0

        # обучаем на батчах
        for data in loader:
            optimizer.zero_grad()
            out = model(data.x, data.edge_index, data.batch)
            loss = criterion(out, data.y)
            total_loss += loss / len(loader)
            acc += accuracy(out.argmax(dim=1), data.y) / len(loader)
            loss.backward()
            optimizer.step()

            # валидируем
            val_loss, val_acc = test(model, val_loader)
```

8. Через каждые 20 эпох мы выводим значения функции потерь и правильности для обучающего и валидационного наборов и в итоге возвращаем обученную модель:

```
        # печатаем метрики через каждые 20 эпох
        if(epoch % 20 == 0):
            print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. наборе: '
                  f'{total_loss:.2f} | Правильность на обуч. наборе: '
                  f'{acc*100:>5.2f}% \n| Функция потерь на валид. '
                  f'наборе: {val_loss:.2f} | Правильность на валид. '
                  f'наборе: {val_acc*100:.2f}%')

    return model
```

9. Функция `test()` также должна включать мини-батчинг, поскольку наши загрузчики валидационных и тестовых данных содержат больше одного батча:

```
@torch.no_grad()
def test(model, loader):
    criterion = torch.nn.CrossEntropyLoss()
    model.eval()
    loss = 0
    acc = 0

    for data in loader:
        out = model(data.x, data.edge_index, data.batch)
        loss += criterion(out, data.y) / len(loader)
        acc += accuracy(out.argmax(dim=1), data.y) / len(loader)

    return loss, acc
```

10. Затем реализуем функцию вычисления правильности:

```
def accuracy(pred_y, y):
    """
```

```
    Вычисляем правильность.
    """
    return ((pred_y == y).sum() / len(y)).item()
```

11. Обучаем модели GCN и GIN и оцениваем их качество:

```
gcn = GCN(dim_h=32)
gcn = train(gcn, train_loader)
test_loss, test_acc = test(gcn, test_loader)
print(f'Функция потерь на тестовом наборе: {test_loss:.2f} | '
      f'Правильность на тестовом наборе: {test_acc*100:.2f}%')
print()

gin = GIN(dim_h=32)
gin = train(gin, train_loader)
test_loss, test_acc = test(gin, test_loader)
print(f'Функция потерь на тестовом наборе: {test_loss:.2f} | '
      f'Правильность на тестовом наборе: {test_acc*100:.2f}%')
```

```
Эпоха   0:
| Функция потерь на обуч. наборе: 0.67 | Правильность на обуч. наборе: 59.80%
| Функция потерь на валид. наборе: 0.64 | Правильность на валид. наборе: 59.47%
Эпоха  20:
| Функция потерь на обуч. наборе: 0.61 | Правильность на обуч. наборе: 70.75%
| Функция потерь на валид. наборе: 0.57 | Правильность на валид. наборе: 76.36%
Эпоха  40:
| Функция потерь на обуч. наборе: 0.61 | Правильность на обуч. наборе: 70.14%
| Функция потерь на валид. наборе: 0.57 | Правильность на валид. наборе: 75.23%
Эпоха  60:
| Функция потерь на обуч. наборе: 0.60 | Правильность на обуч. наборе: 71.56%
| Функция потерь на валид. наборе: 0.57 | Правильность на валид. наборе: 75.52%
Эпоха  80:
| Функция потерь на обуч. наборе: 0.60 | Правильность на обуч. наборе: 69.33%
| Функция потерь на валид. наборе: 0.55 | Правильность на валид. наборе: 75.23%
Эпоха 100:
| Функция потерь на обуч. наборе: 0.60 | Правильность на обуч. наборе: 70.32%
| Функция потерь на валид. наборе: 0.55 | Правильность на валид. наборе: 78.49%
Функция потерь на тестовом наборе: 0.60 | Правильность на тестовом наборе: 75.00%

Эпоха   0:
| Функция потерь на обуч. наборе: 1.54 | Правильность на обуч. наборе: 57.90%
| Функция потерь на валид. наборе: 0.57 | Правильность на валид. наборе: 64.51%
Эпоха  20:
| Функция потерь на обуч. наборе: 0.53 | Правильность на обуч. наборе: 74.78%
| Функция потерь на валид. наборе: 0.49 | Правильность на валид. наборе: 82.40%
Эпоха  40:
| Функция потерь на обуч. наборе: 0.51 | Правильность на обуч. наборе: 75.50%
| Функция потерь на валид. наборе: 0.47 | Правильность на валид. наборе: 79.21%
Эпоха  60:
| Функция потерь на обуч. наборе: 0.50 | Правильность на обуч. наборе: 75.67%
| Функция потерь на валид. наборе: 0.45 | Правильность на валид. наборе: 76.30%
```

```
Эпоха  80:
| Функция потерь на обуч. наборе: 0.51 | Правильность на обуч. наборе: 74.63%
| Функция потерь на валид. наборе: 0.52 | Правильность на валид. наборе: 80.83%
Эпоха 100:
| Функция потерь на обуч. наборе: 0.50 | Правильность на обуч. наборе: 74.26%
| Функция потерь на валид. наборе: 0.53 | Правильность на валид. наборе: 78.71%
Функция потерь на тестовом наборе: 0.56 | Правильность на тестовом наборе: 72.14%
```

Мы можем заключить, что для этой задачи классификации графов архитектура GCN сработала чуть лучше, чем архитектура GIN. В целом считается, что GCN менее выразительны, чем GIN. Другими словами, GIN могут различать больше графовых структур, чем GCN, поэтому они более точны. Теперь визуализируем ошибки, допущенные обеими моделями.

1. Сначала нам нужно импортировать библиотеки matplotlib и networkx для построения графов белков 4×4:

```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from torch_geometric.utils import to_networkx
```

2. Затем мы берем результаты классификации, если прогноз является верным, визуализируем граф с помощью зеленого цвета, в противном случае используем красный цвет. Для удобства используем преобразование в граф библиотеки networkx. Потом мы можем визуализировать его с помощью функции `nx.draw_networkx()`:

```
fig, ax = plt.subplots(4, 4)
fig.suptitle('GCN - Классификация графов')

for i, data in enumerate(dataset[-16:]):
    # вычисляем цвет (зеленый, если правильно,
    # красный в противном случае)
    out = gcn(data.x, data.edge_index, data.batch)
    color = "green" if out.argmax(dim=1) == data.y else "red"

    # визуализируем графы
    ix = np.unravel_index(i, ax.shape)
    ax[ix].axis('off')
    G = to_networkx(dataset[i], to_undirected=True)
    nx.draw_networkx(G,
                     pos=nx.spring_layout(G, seed=0),
                     with_labels=False,
                     node_size=10,
                     node_color=color,
                     width=0.8,
                     ax=ax[ix]
                    )
```

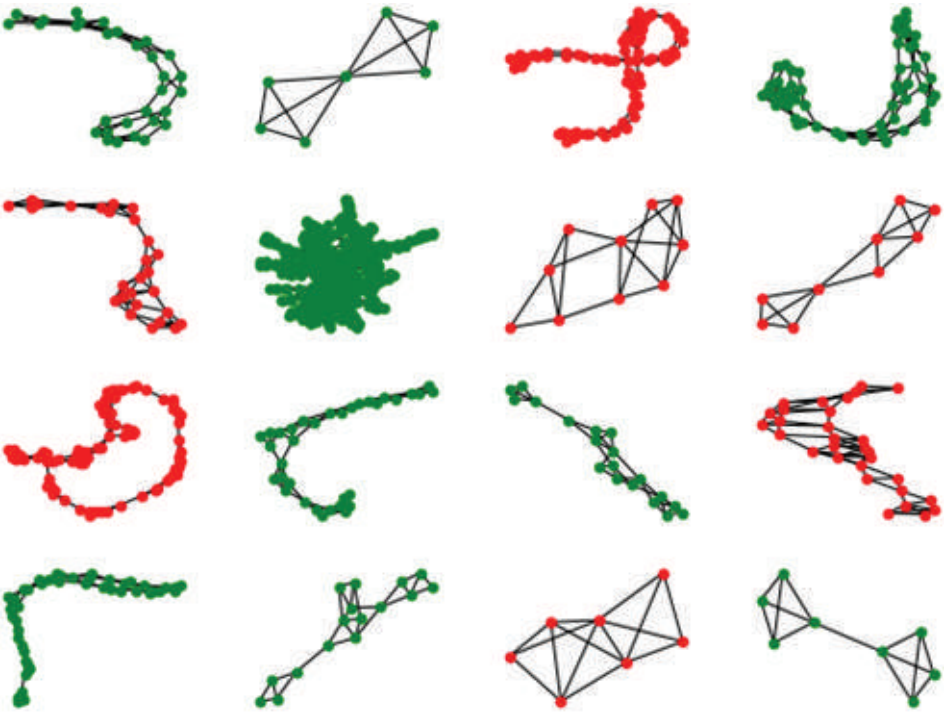


Рис. 9.5 ❖ Результаты классификации графов, полученные с помощью модели GCN

```
fig, ax = plt.subplots(4, 4)
fig.suptitle('GIN - Классификация графов')

for i, data in enumerate(dataset[-16:]):
    # вычисляем цвет (зеленый, если правильно,
    # красный в противном случае)
    out = gin(data.x, data.edge_index, data.batch)
    color = "green" if out.argmax(dim=1) == data.y else "red"

    # визуализируем графы
    ix = np.unravel_index(i, ax.shape)
    ax[ix].axis('off')
    G = to_networkx(dataset[i], to_undirected=True)
    nx.draw_networkx(G,
                     pos=nx.spring_layout(G, seed=0),
                     with_labels=False,
                     node_size=10,
                     node_color=color,
                     width=0.8,
                     ax=ax[ix]
                    )
```

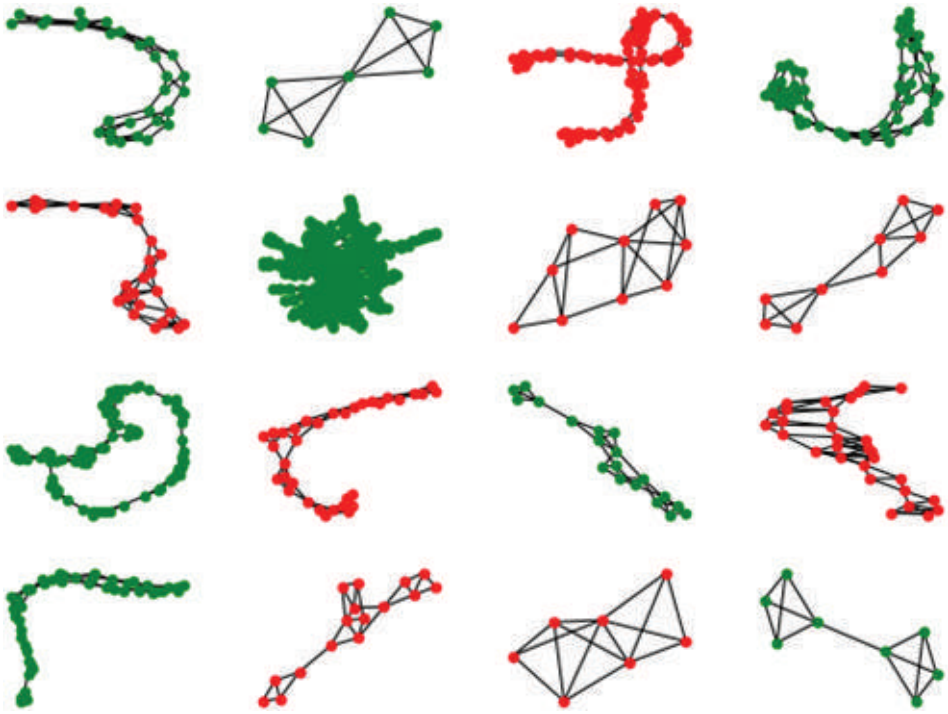


Рис. 9.6 ❖ Результаты классификации графов, полученные с помощью модели GIN

Видим, что модель GIN допускает чуть больше ошибок. Понимание того, какие графовые структуры не фиксируются должным образом, потребует обширного анализа каждого белка, правильно классифицированного GCN. Часто бывает так, что одна модель хорошо классифицирует одни графы, а другая модель хорошо классифицирует другие графы. В таком случае модели могут дополнять друг друга.

Создание ансамблей из моделей, неправильно классифицирующих разные графы, – распространенный метод машинного обучения. Мы могли бы использовать разные подходы, например взять третью модель, обученную на наших итоговых результатах классификации. Поскольку создание ансамблей не является целью этой главы, мы реализуем простой метод усреднения моделей.

1. Сначала задаем для моделей режим оценки качества (тестовый режим) и определяем переменные для хранения значений правильности:

```
# задаем режим оценки
gcn.eval()
gin.eval()

# сюда записываем значения правильности
acc_gcn = 0
```

```
acc_gin = 0
acc_ens = 0
```

2. Получаем результаты классификации по каждой модели и комбинируем для получения прогнозов ансамбля:

```
for data in test_loader:
    # получаем результаты классификации
    out_gcn = gcn(data.x, data.edge_index, data.batch)
    out_gin = gin(data.x, data.edge_index, data.batch)
    out_ens = (out_gcn + out_gin)/2
```

3. Вычисляем значения правильности для трех наборов прогнозов:

```
    # вычисляем значения правильности
    acc_gcn += accuracy(out_gcn.argmax(dim=1), data.y) / len(test_loader)
    acc_gin += accuracy(out_gin.argmax(dim=1), data.y) / len(test_loader)
    acc_ens += accuracy(out_ens.argmax(dim=1), data.y) / len(test_loader)
```

4. Печатаем результаты:

```
# печатаем результаты
print(f'GCN правильность:     {acc_gcn*100:.2f}%')
print(f'GIN правильность:     {acc_gin*100:.2f}%')
print(f'GCN+GIN правильность: {acc_ens*100:.2f}%')

GCN правильность:     74.22%
GIN правильность:     70.31%
GCN+GIN правильность: 70.83%
```

В этом примере наш ансамбль работает на уровне модели GIN, но хуже, чем модель GCN. Тем не менее этот результат важен, поскольку показывает возможности, предлагаемые этим методом. Как вариант, мы могли бы дополнить ансамбль эмбеддингами из других архитектур, таких как Node2Vec, и посмотреть, улучшит ли это итоговую правильность.

Выводы

В этой главе мы убедились в выразительной силе графовой сети изоморфизма (GNN). Она опирается на другой алгоритм – метод WL, который создает каноническую форму графа. Этот алгоритм не идеален, но он может различать большое количество графовых структур. Он вдохновил исследователей на создание архитектуры GIN, призванной быть такой же выразительной, как тест WL, и, следовательно, строго более выразительной, чем GCN, GAT или GraphSAGE.

Затем мы реализовали эту архитектуру для классификации графов. Мы рассмотрели различные методы объединения эмбеддингов узлов в эмбеддинги графов. GIN предлагает новый метод, который включает в себя агре-

гатор на основе суммы и конкатенацию эмбеддингов графов, созданных каждым GIN-слоем. Он значительно превосходит классический глобальный пулинг на основе среднего, получаемый с помощью GCN-слоев. Наконец, мы объединили прогнозы, полученный обеими моделями, в простой ансамбль, чтобы попытаться улучшить правильность.

В главе 10 «Прогнозирование связей с помощью графовых нейронных сетей» мы рассмотрим еще одну популярную задачу с использованием GNN – прогнозирование связей. На самом деле этот подход не совсем является новым, поскольку предыдущие методы, которые мы видели, такие как DeepWalk и Node2Vec, уже были основаны на этой идее. Мы объясним, почему он не новый, и представим два новых инструмента на основе GNN – графовый (вариационный) автоэнкодер и фреймворк SEAL. Наконец, мы реализуем их и сравним качество их прогнозов, решая задачу прогнозирования связей на примере набора данных `Cora`.

Дополнительное чтение

[1] Weisfeiler and Lehman, A.A. (1968) A Reduction of a Graph to a Canonical Form and an Algebra Arising during This Reduction. Nauchno-Technicheskaya Informatsia, 9.

[2] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, *How Powerful are Graph Neural Networks?* arXiv, 2018. doi: 10.48550/ARXIV.1810.00826.

[3] C. Morris et al., *Weisfeiler and Leman Go Neural: Higher-order Graph Neural Networks*. arXiv, 2018. doi: 10.48550/ARXIV.1810.02244.

[4] V. P. Dwivedi et al. *Benchmarking graph neural networks*. arXiv, 2020. doi: 10.48550/ARXIV.2003.00982.

[5] K. M. Borgwardt, C. S. Ong, S. Schoenauer, S. V. N. Vishwanathan, A. J. Smola, and H. P. Kriegel. *Protein function prediction via graph kernels*. Bioinformatics, 21 (Suppl 1): i47–i56, Jun 2005.

[6] P. D. Dobson and A. J. Doig. *Distinguishing enzyme structures from non-enzymes without alignments*. J. Mol. Biol., 330 (4): 771–783, Jul 2003.

[7] Christopher Morris and Nils M. Kriege and Franka Bause and Kristian Kersting and Petra Mutzel and Marion Neumann. *TUDataset: A collection of benchmark datasets for learning with graphs*. In ICML 2020 Workshop on Graph Representation Learning and Beyond.

[8] W. Hu et al., *Strategies for Pre-training Graph Neural Networks*. arXiv, 2019. doi: 10.48550/ARXIV.1905.12265.

Глава 10

Прогнозирование связей с помощью графовых нейронных сетей

Прогнозирование связей, или **прогнозирование появления ребер** (link prediction), – одна из самых популярных задач, решаемых с помощью графов. Мы прогнозируем вероятность наличия связи (ребра) между двумя узлами. Задача актуальна для социальных сетей и рекомендательных систем. Хорошим примером является отображение общих друзей в социальных сетях. *Общие друзья* – это пользователи, имеющие как минимум одного друга, который также является вашим другом. Интуитивно понятно, что у вас больше шансов завести знакомство с этими людьми. Именно эту вероятность мы и пытается оценить в рамках прогнозирования связей.

В этой главе мы сначала выясним, как выполнить прогнозирование связей, не прибегая к какому-либо машинному обучению. Эти традиционные методы необходимы для понимания того, что исследуют GNN. Затем мы обратимся к предыдущим главам, посвященным DeepWalk и Node2Vec, чтобы проиллюстрировать прогнозирование связей с помощью **матричной факторизации** (matrix factorization). К сожалению, эти методы имеют существенные ограничения, поэтому мы перейдем к методам на основе GNN.

Рассмотрим три метода из двух разных семейств. Первое семейство использует эмбеддинги узлов и выполняет матричную факторизацию матрицы на основе GNN. Второй метод использует представление подграфа. Окрестности каждой связи (фиктивной или фактической) считаются входными данными для прогнозирования вероятности связи. Наконец, мы реализуем модель каждого семейства в PyTorch Geometric.

К концу этой главы вы сможете реализовывать различные методы прогнозирования связей. Решая задачу прогнозирования связей, вы будете знать, какой метод наилучшим образом подходит в вашем случае – эвристика, матричная факторизация, эмбеддинги на основе GNN или техники на основе подграфов.

В этой главе мы рассмотрим следующие основные темы:

- «Прогнозирование связей с помощью традиционных методов»,
- «Прогнозирование связей с помощью эмбеддингов узлов»,
- «Прогнозирование связей с помощью SEAL».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter10.

Прогнозирование связей с помощью традиционных методов

Проблема прогнозирования связей существует уже долгое время, поэтому было предложено множество методов для ее решения. Сначала в данном разделе мы опишем популярные эвристики, основанные на локальных и глобальных окрестностях. Затем представим факторизацию матриц и ее связь с методами DeepWalk и Node2Vec.

Эвристические методы

Эвристические методы представляют собой простой и практический способ прогнозирования связей между узлами. Они легки в реализации и являются базовыми моделями для решения этой задачи. Мы можем классифицировать их в зависимости от количества уровней, с которым они работают (см. рис. 10.1). Некоторым для работы требуются только соседние узлы 1-го уровня, смежные с целевыми узлами. Более сложные методы дополнительно учитывают соседние узлы 2-го уровня или весь граф. В этом разделе мы разделим эвристики на две категории – локальные (работают с соседними узлами 1-го и 2-го уровней) и глобальные.

Локальные эвристики измеряют схожесть между двумя узлами, рассматривая их локальные окрестности. Мы используем $\mathcal{N}(u)$ для обозначения соседей узла u. Давайте разберем три популярные локальные эвристики.

- **Общие соседи** (common neighbors): просто подсчитываем количество общих соседей двух узлов (количество соседних узлов 1-го уровня). Идея аналогична нашему предыдущему примеру с социальными сетями – чем больше общих соседей, тем больше вероятность, что узлы соединены друг с другом:

$$f(u, v) = \{\mathcal{N}(u) \cap \mathcal{N}(v)|.$$

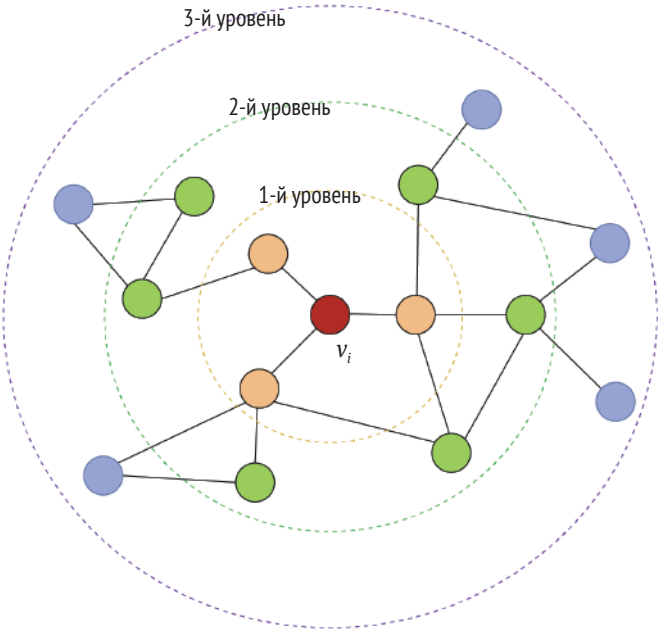


Рис. 10.1 ❖ Граф с соседними узлами 1-го, 2-го и 3-го уровней

- **Коэффициент Жаккара** (Jaccard's coefficient): измеряет долю общих соседей двух узлов (соседних узлов 1-го уровня). Эвристика основывается на той же идее, что и общие соседи, но нормирует результат по общему количеству соседей. Она поощряет узлы с небольшим количеством взаимосвязанных соседей вместо узлов с высокими степенями (узлов, которые имеют большое количество соседей):

$$f(u,v) = \frac{|\mathcal{N}(u) \cap \mathcal{N}(v)|}{|\mathcal{N}(u) \cup \mathcal{N}(v)|}.$$

- **Индекс Адамика–Адара** (Adamic-Adar index): представляет собой сумму обратных логарифмических степеней общих соседей для двух целевых узлов (соседних узлов 2-го уровня). Идея заключается в том, что общие соседи с большими окрестностями менее важны, чем те, у которых окрестности меньше. Именно поэтому они должны иметь меньшее значение в конечном результате:

$$f(u,v) = \sum_{x \in \mathcal{N}(u) \cap \mathcal{N}(v)} \frac{1}{\log|\mathcal{N}(x)|}.$$

Все эти методы основываются на степенях соседних узлов, будь то прямые (общие соседи или коэффициент Жаккара) или косвенные соседи (индекс Адамика–Адара). Это важно для скорости работы и интерпретируемости, но также ограничивает сложность связей, которые они могут улавливать.

Глобальные эвристики предлагают решение этой проблемы, рассматривая не локальную окрестность, а всю сеть. Приведем две хорошо известные глобальные эвристики.

- **Индекс Каца** (Katz index) вычисляет взвешенную сумму каждого возможного пути между двумя узлами. Веса соответствуют дисконт-фактору $\beta \in [0, 1]$ (обычно между 0.8 и 0.9), чтобы штрафовать более длинные пути. Согласно такому определению два узла с большей вероятностью будут соединены, если между ними существует много (желательно коротких) путей. Пути любой длины можно вычислить путем возведения матрицы смежности в степень A^n, поэтому индекс Каца определяется следующим образом:

$$f(u,v) = \sum_{i=1}^{\infty} \beta^i A^i.$$

- **Случайное блуждание с перезапуском** (random walk with restart) [1] выполняет случайные блуждания, начиная с целевого узла. После каждого блуждания увеличивается счетчик посещения текущего узла. С вероятностью α алгоритм перезапускает блуждание в целевом узле. В противном случае он продолжает случайное блуждание. После заранее определенного количества итераций мы останавливаем алгоритм и можем предложить связи между целевым узлом и узлами с наибольшим числом посещений. Эта идея также является важной в алгоритмах DeepWalk и Node2Vec.

Глобальные эвристики обычно более точны, но требуют наличия информации о всем графе. Однако это не единственный способ прогнозирования связей с помощью такой информации.

Матричная факторизация

Метод матричной факторизации для прогнозирования связей вдохновлен ранними исследованиями в области рекомендательных систем [2]. С помощью этого метода мы прогнозируем связи косвенно, прогнозируя всю матрицу смежности $\hat{A}$. Это достигается с помощью эмбеддингов узлов – схожие узлы u и v должны иметь схожие эмбеддинги узлов z_u и z_v соответственно. Используя скалярное произведение, мы можем записать это утверждение следующим образом:

- если эти узлы похожи, то скалярное произведение эмбеддингов $z_v^T z_u$ должно быть максимальным;
- если эти узлы не похожи, то скалярное произведение эмбеддингов $z_v^T z_u$ должно быть минимальным.

До сих пор мы предполагали, что похожие друг на друга узлы должны быть связаны между собой. Вот почему мы можем использовать это скалярное

произведение для аппроксимации каждого элемента (связи) матрицы смежности A:

$$A_{uv} \approx z_v^T z_u.$$

В терминах умножения матриц мы получаем следующее выражение:

$$A \approx Z^T Z,$$

где Z – это матрица эмбеддингов узлов. На рис. 10.2 приведено визуальное объяснение матричной факторизации:

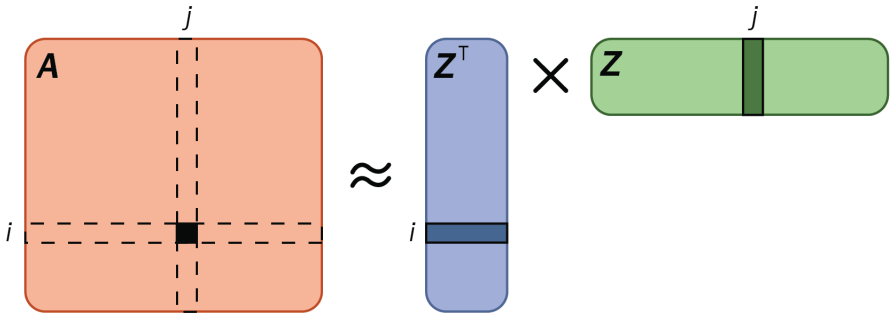


Рис. 10.2 ❖ Матричная факторизация с помощью эмбеддингов узлов

Этот метод называется матричная факторизация, потому что матрицу смежности можно разложить, т. е. представить в виде произведения двух матриц. Задача заключается в том, чтобы извлечь релевантные эмбеддинги узлов, которые минимизируют L2-норму между фактическими и предсказанными элементами матрицы смежности A_{uv} для графа $G = (V, E)$:

$$\underset{z}{\text{minimize}} \sum_{i \in V,\, j \in V} (A_{uv} - z_v^T z_u)^2.$$

Существуют более сложные варианты матричной факторизации, которые включают матрицу Лапласа и степени. Альтернативным решением является использование моделей типа DeepWalk и Node2Vec. Они создают эмбеддинги узлов, которые можно объединить для создания представлений связей. Согласно Цю и соавторам [3], эти алгоритмы неявно аппроксимируют и факторизуют сложные матрицы. Например, вот матрица, вычисленная с помощью DeepWalk:

$$\log\left(\sum_{i=1}^{|V|}\sum_{j=1}^{|V|} A_{i,j}\left(\frac{1}{T}\sum_{r=1}^{T}(D^{-1}A)^r\right)D^{-1}\right)\log b.$$

Здесь b – это параметр отрицательного семплирования. То же самое можно сказать и о похожих алгоритмах типа LINE и PTE. Хотя они могут улавливать

более сложные зависимости, они сталкиваются с теми же ограничениями, которые мы видели в главах 3 и 4:

- **не могут использовать признаки узлов**: они используют только топологическую информацию для создания эмбеддингов;
- **у них нет индуктивных возможностей**: они не могут обобщать на узлы, которых не было в обучающем наборе;
- **они не могут улавливать структурное сходство**: узлы графа, структурно похожие друг на друга, могут получать сильно различающиеся эмбеддинги.

Эти ограничения мотивируют необходимость в методах, основанных на графовых нейронных сетях (GNN), как мы увидим в следующих разделах.

Прогнозирование связей с помощью эмбеддингов узлов

В предыдущих главах мы узнали, как использовать GNN для создания эмбеддингов узлов. Популярной техникой прогнозирования связей является использование этих эмбеддингов для выполнения матричной факторизации. В этом разделе мы рассмотрим две архитектуры GNN для прогнозирования связей – **графовый автоэнкодер** (Graph Autoencoder – GAE) и **вариационный графовый автоэнкодер** (Variational Graph Autoencoder – VGAE).

Знакомство с графовыми автоэнкодерами

Обе архитектуры были представлены Кипфом и Веллингом в 2016 году [5] в статье объемом 3 страницы. Они представляют собой графовые нейронные сети, аналогичные двум популярным архитектурам нейронных сетей – автоэнкодеру и вариационному автоэнкодеру. Предварительные знания об этих архитектурах полезны, но не обязательны. Для легкости понимания мы сначала сосредоточимся на графовом автоэнкодере.

Графовый автоэнкодер состоит из двух модулей.

- **Кодировщик**, или **энкодер** (encoder), – это классическая двухслойная графовая сверточная нейронная сеть (GCN), которая вычисляет эмбеддинги узлов следующим образом:

$$Z = GCN(X, A).$$

- **Декодировщик**, или **декодер** (decoder), аппроксимирует матрицу смежности $\hat{A}$ с помощью матричной факторизации и сигмоидной функции σ для вывода вероятностей:

$$\hat{A} = \sigma(Z^T Z).$$

Обратите внимание, что мы не пытаемся классифицировать узлы или графы. Цель – спрогнозировать вероятность (в диапазоне между 0 и 1) для каждого элемента матрицы смежности $\hat{A}$. Поэтому графовый автоэнкодер обучается с помощью бинарной кросс-энтропии (отрицательной логарифмической функции правдоподобия), в основе которой лежит сравнение фактических и предсказанных значений матрицы смежности:

$$\mathcal{L}_{\text{BCE}} = \sum_{i \in V,\, j \in V} -A_{ij} \log(\hat{A}_{ij}) - (1 - A_{ij}) \log(1 - \hat{A}_{ij}).$$

Однако матрицы смежности часто бывают очень разреженными, что смещает графовый автоэнкодер в сторону предсказания нулевых значений. Есть две простые техники для устранения этого смещения. Во-первых, мы можем добавить вес в пользу $A_{ij} = 1$ в вышеприведенной функции потерь. Во-вторых, мы можем отбирать меньше нулевых значений во время обучения, чтобы сделать прогнозы более сбалансированными. Последняя техника реализована Кипфом и Веллингом.

Эта архитектура является гибкой – кодировщик можно заменить другим типом графовой нейронной сети (например, GraphSAGE), а декодировщиком может быть, например, MLP. Еще одно возможное улучшение подразумевает вероятностную реализацию графового автоэнкодера – вариационный автоэнкодер.

Знакомство с вариационными графовыми автоэнкодерами

Различие между графовыми автоэнкодерами и вариационными графовыми автоэнкодерами аналогично различию между автоэнкодерами и вариационными автоэнкодерами. Вместо непосредственного выучивания эмбеддингов узлов вариационные графовые автоэнкодеры выучивают параметры нормальных распределений, которые затем используются для создания эмбеддингов. Вариационные графовые автоэнкодеры также разделены на два модуля:

- **кодировщик**, или **энкодер** (encoder), обычно состоит из трех GCN-слоев: общего слоя и двух слоев для аппроксимации параметров распределения. Цель заключается в том, чтобы выучить параметры каждого скрытого нормального распределения – среднее μ_i (выучиваем с помощью GCN_μ-слоя) и дисперсию σ_i^2 (на практике логарифм стандартного отклонения, выучиваем с помощью GCN_σ-слоя);
- **декодировщик**, или **декодер** (decoder), семплирует эмбеддинги z_i из выученных распределений с помощью трюка перепараметризации [4]. Затем он использует то же самое скалярное произведение скрытых переменных для аппроксимации матрицы смежности $\hat{A} = \sigma(Z^T Z)$.

При работе с VGAE важно обеспечить, чтобы вывод кодировщика подчинялся нормальному распределению. Поэтому мы добавляем новый член в функцию потерь – дивергенцию Кульбака–Лейблера (KL), которая измеряет расхождение между двумя распределениями. Мы получаем следующую функцию потерь, еще называемую **нижней границей функции логарифмического правдоподобия** или **нижней вариационной границей** (evidence lower bound – ELBO):

$$\mathcal{L}_{ELBO} = \mathcal{L}_{BCE} - KL[q(Z|X, A)||p(Z)].$$

Здесь $q(Z|X, A)$ представляет собой кодировщик, а $p(Z)$ – априорное распределение Z.

Качество модели обычно оценивается с помощью двух метрик – **площади** под ROC-кривой (AUC-ROC) и **средней точности** (average precision – AP). Давайте посмотрим, как реализовать VGAE с помощью PyTorch Geometric.

Реализация VGAE

Здесь у нас будут два основных отличия от предыдущих реализаций графовых нейронных сетей.

- Мы предварительно обработаем набор данных.
- Мы создадим модель кодировщика, которую передадим в класс `VGAE`, вместо того чтобы напрямую реализовывать VGAE с нуля. Нижеприведенный программный код вдохновлен примером VGAE из PyTorch Geometric.

1. Зададим некоторые настройки и напишем собственную функцию для обеспечения воспроизводимости результатов:

```
import torch
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

def set_seed():
    """
    Задает стартовое значение генератора псевдослучайных
    чисел для воспроизводимости.
    """
    torch.manual_seed(-1)
    torch.cuda.manual_seed(0)
    torch.cuda.manual_seed_all(0)
```

2. Импортируем необходимые библиотеки:

```
import numpy as np
import matplotlib.pyplot as plt
import torch_geometric.transforms as T
from torch_geometric.datasets import Planetoid
```

3. При наличии возможности воспользуемся GPU:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

4. Мы создаем объект `transform`, который нормализует входные признаки, непосредственно выполняет преобразование тензора в соответствии с выбранным устройством и случайным образом разделяет связи. В данном примере мы используем разделение 85/5/10. Для параметра `add_negative_train_samples` задаем значение `False`, потому что модель уже выполняет отрицательное семплирование, поэтому оно не требуется для набора данных:

```
set_seed()

transform = T.Compose([
    T.NormalizeFeatures(),
    T.ToDevice(device),
    T.RandomLinkSplit(num_val=0.05,
                      num_test=0.1,
                      is_undirected=True,
                      split_labels=True,
                      add_negative_train_samples=False)
])
```

5. Мы загружаем набор данных `Cora`, используя ранее созданный объект `transform`:

```
dataset = Planetoid('.', name='Cora', transform=transform)
```

6. Процедура `RandomLinkSplit` позволяет нам получить обучающий, валидационный и тестовый наборы, используя случайное разбиение графовых данных на уровне связей:

```
train_data, val_data, test_data = dataset[0]
```

7. Теперь реализуем кодировщик. Сначала нужно импортировать классы `GCNConv` и `VGAE`:

```
from torch_geometric.nn import GCNConv, VGAE
```

8. Теперь пишем свой класс-кодировщик. В этом классе нам нужно реализовать три GCN-слоя: общий слой, второй слой для аппроксимации средних значений μ_i, третий слой для аппроксимации значений дисперсии σ_i^2 (на практике аппроксимируют логарифм стандартного отклонения):

```
class Encoder(torch.nn.Module):
    def __init__(self, dim_in, dim_out):
        super().__init__()
        self.conv1 = GCNConv(dim_in, 2 * dim_out)
        self.conv_mu = GCNConv(2 * dim_out, dim_out)
        self.conv_logstd = GCNConv(2 * dim_out, dim_out)
```

```
    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        return self.conv_mu(x, edge_index), self.conv_logstd(x, edge_index)
```

9. Теперь создаем экземпляр класса `VGAE`, чтобы передать наш класс-кодировщик в качестве входных данных. По умолчанию он будет использовать скалярное произведение в качестве декодировщика:

```
model = VGAE(Encoder(dataset.num_features, 16)).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

10. Функция `train()` включает в себя два важных шага. Сначала вычисляется матрица эмбеддингов с помощью `model.encode()`, название может показаться контринтуитивным, но эта функция действительно генерирует эмбеллинги, выбранные из выученных распределений. Затем вычисляется функция потерь ELBO с помощью функций `model.recon_loss()` (бинарная кросс-энтропия) и `model.kl_loss()` (дивергенция Кульбака–Лейблера). Декодировщик вызывается неявно для вычисления кросс-энтропии.

```
set_seed()

def train():
    model.train()
    optimizer.zero_grad()
    z = model.encode(train_data.x, train_data.edge_index)
    loss = model.recon_loss(z, train_data.pos_edge_label_index) + (
        1 / train_data.num_nodes) * model.kl_loss()
    loss.backward()
    optimizer.step()
    return float(loss)
```

11. Функция `test()` просто вызывает соответствующий метод класса `VGAE`:

```
@torch.no_grad()
def test(data):
    model.eval()
    z = model.encode(data.x, data.edge_index)
    return model.test(z,
                      data.pos_edge_label_index,
                      data.neg_edge_label_index)
```

12. Мы обучаем модель, задав 301 эпоху, и печатаем метрики AUC-ROC и AP:

```
for epoch in range(301):
    loss = train()
    val_auc, val_ap = test(test_data)
    if epoch % 50 == 0:
        print(f'Эпоха {epoch:>2}:\n| Функция потерь: {loss:.4f} | '
              f'AUC-ROC на валид. наборе {val_auc:.4f} | '
```

```
                f'AP на валид. наборе: {val_ap:.4f}%')
```

```
Эпоха  0:
| Функция потерь: 3.5054 | AUC-ROC на валид. наборе 0.7005 | AP на валид. наборе:
0.7225%
Эпоха 50:
| Функция потерь: 1.3234 | AUC-ROC на валид. наборе 0.6720 | AP на валид. наборе:
0.7035%
Эпоха 100:
| Функция потерь: 1.1940 | AUC-ROC на валид. наборе 0.7452 | AP на валид. наборе:
0.7523%
Эпоха 150:
| Функция потерь: 1.0715 | AUC-ROC на валид. наборе 0.8080 | AP на валид. наборе:
0.8148%
Эпоха 200:
| Функция потерь: 0.9864 | AUC-ROC на валид. наборе 0.8623 | AP на валид. наборе:
0.8558%
Эпоха 250:
| Функция потерь: 0.9437 | AUC-ROC на валид. наборе 0.8697 | AP на валид. наборе:
0.8643%
Эпоха 300:
| Функция потерь: 0.9279 | AUC-ROC на валид. наборе 0.8831 | AP на валид. наборе:
0.8791%
```

13. Оцениваем качество модели на тестовом наборе:

```
test_auc, test_ap = test(test_data)
print(f'AUC-ROC на тестовом наборе: {test_auc:.4f} | '
      f'AP на тестовом наборе {test_ap:.4f}')
```

```
AUC-ROC на тестовом наборе: 0.8831 | AP на тестовом наборе 0.8791
```

14. В итоге вручную вычисляем аппроксимированную матрицу смежности $\hat{A}$:

```
z = model.encode(test_data.x, test_data.edge_index)
Ahat = torch.sigmoid(z @ z.T)
Ahat
```

```
tensor([[0.7775, 0.6279, 0.7154,  ..., 0.6792, 0.7742, 0.7751],
        [0.6279, 0.8846, 0.8908,  ..., 0.5759, 0.8445, 0.8181],
        [0.7154, 0.8908, 0.9154,  ..., 0.6002, 0.8848, 0.8681],
        ...,
        [0.6792, 0.5759, 0.6002,  ..., 0.6686, 0.6677, 0.6633],
        [0.7742, 0.8445, 0.8848,  ..., 0.6677, 0.8789, 0.8663],
        [0.7751, 0.8181, 0.8681,  ..., 0.6633, 0.8663, 0.8572]],
       grad_fn=<SigmoidBackward0>)
```

Обучение VGAE происходит быстро, и его результаты легко интерпретировать. Однако нередки случаи, когда GCN не хватает выразительности. Для улучшения выразительности модели нам нужно реализовать более эффективные методы.

Прогнозирование связей с помощью SEAL

В предыдущем разделе мы познакомились с методами, которые используют узлы графа и выучивают соответствующие эмбеддинги узлов для вычисления вероятностей связей. Еще один метод заключается в рассмотрении локальных окрестностей вокруг целевых узлов. Эти методы называются алгоритмами на основе подграфов и были популяризированы в рамках фреймворка **SEAL** (что можно расшифровать как **Subgraphs, Embeddings, and Attributes for Link prediction** – подграфы, эмбеддинги и атрибуты для прогнозирования связей). В данном разделе мы рассмотрим фреймворк SEAL и реализуем его с использованием PyTorch Geometric.

Знакомство с фреймворком SEAL

Предложенный в 2018 году Чжаном и Чэном [6], SEAL – это фреймворк, который исследует структурные признаки графа для прогнозирования связей. Он определяет подграф, образованный целевыми узлами (x, y) и их соседями k-го уровня, в виде **охватывающего подграфа** (enclosing subgraph). Каждый охватывающий подграф используется в качестве входных данных (вместо всего графа) для прогнозирования вероятности связи. SEAL автоматически выучивает локальную эвристику для прогнозирования связи.

Фреймворк включает три этапа.

1. **Извлечение охватывающего подграфа** (enclosing subgraph extraction), которое включает в себя формирование набора реальных связей и набора фиктивных связей (отрицательное семплирование) для получения обучающих данных.
2. **Построение матрицы с информацией об узлах** (node information matrix construction), которое включает в себя три компонента – метки узлов, эмбеддинги узлов и признаки узлов.
3. **Обучение графовых нейронных сетей** (GNN training), которое принимает на вход матрицы с информацией об узлах и выдает вероятности связей.

Эти шаги можно подытожить на схеме, представленной на рис. 10.3.

Извлечение охватывающего подграфа – это простой процесс. Он включает в себя перечисление целевых узлов и их соседей k-го уровня для извлечения связей и признаков. Высокое значение k улучшит качество эвристик, которые SEAL может выучить, но также создаст более крупные подграфы, требующие больше вычислительных ресурсов.

Первый этап извлечения информации об узле – это разметка узла. В рамках этой процедуры каждому узлу присваивается конкретный номер. Без него GNN не сможет отличать целевые узлы от контекстных узлов (соседей

целевых узлов). Также вводятся расстояния, описывающие относительные положения узлов и их структурную важность.

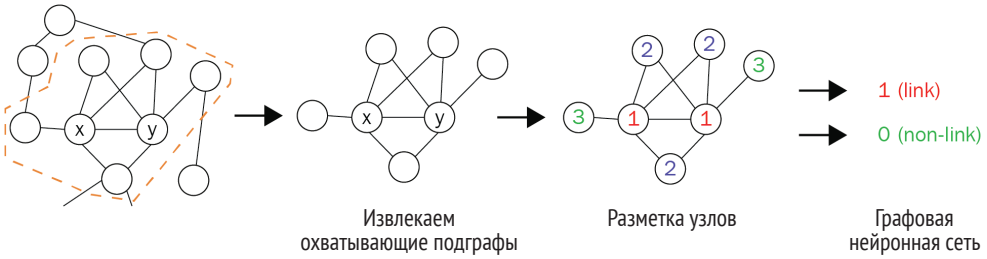


Рис. 10.3 ❖ Фреймворк SEAL

На практике целевые узлы *x* и *y* должны иметь общую уникальную метку, чтобы их можно было идентифицировать как целевые узлы. У контекстных узлов *i* и *j*должна быть одинаковая метка, если они находятся на одинаковом расстоянии от целевых узлов, т. е. $d(i, x) = d(j, x)$ и $d(i, y) = d(j, y)$. Это расстояние называется двойным радиусом, обозначаемым как $(d(i, x), d(i, y))$.

Существует несколько вариантов решения, но авторы SEAL предлагают алгоритм **разметки узлов на основе двойного радиуса** (Double-Radius Node Labeling – DRNL). Он работает следующим образом.

1. Сначала присваиваем метку 1 для целевых узлов *x* и *y*.
2. Присвоим метку 1 узлам с радиусом (1, 1).
3. Присвоим метку 3 узлам с радиусом (1, 2) или (2, 1).
4. Присвоим метку 4 узлам с радиусом (1, 3), (3, 1) и т. д.

Функцию DRNL можно записать следующим образом:

$$f(i) = 1 + \min(d(i, x), d(i, y)) + (d/2)[(d/2) + (d\%2) - 1].$$

Здесь $d = d(i, x) + d(i, y)$, а $(d/2)$ и $(d\%2)$ – это целая часть и остаток от деления *d* на 2 соответственно. Наконец, эти метки узлов подвергаются one-hot-кодированию.

> **Примечание**
>
> Два других компонента легче получить. Эмбеддинги узлов не обязательны, но их можно вычислить с использованием другого алгоритма типа Node2Vec. Затем они объединяются с признаками узла и метками, подвергнутыми one-hot-кодированию, для построения итоговой матрицы, содержащей информацию об узлах.

Наконец, GNN обучается прогнозировать связи, используя информацию об охватывающих подграфах и матрицах смежности. В этой задаче авторы SEAL выбрали модель **глубокой графовой сверточной нейронной сети** (Deep Graph Convolutional Neural Network – DGCNN). Эта архитектура включает три шага.

1. Несколько GCN-слоев вычисляют эмбеддинги узлов, которые затем конкатенируются (как в GIN).
2. Слой глобальной сортировки (или пулинга) объединяет эти эмбеддинги в согласованном порядке перед передачей их в сверточные слои, которые не инвариантны к перестановкам (т. е. они учитывают структуру данных и сохраняют информацию о порядке элементов).
3. К отсортированным графовым представлениям применяются традиционные сверточные и плотные слои, и выводится вероятность связи.

Модель DGCNN обучается с использованием бинарной кросс-энтропии и выдает вероятности от 0 до 1.

Реализация фреймворка SEAL

Фреймворк SEAL требует обширной предварительной обработки для извлечения и разметки охватывающих подграфов. Давайте реализуем это с помощью PyTorch Geometric.

1. Сначала мы импортируем все необходимые библиотеки, классы и функции:

```
import numpy as np
from sklearn.metrics import (roc_auc_score,
                             average_precision_score)
from scipy.sparse.csgraph import shortest_path

import torch.nn.functional as F
from torch.nn import (Conv1d,
                      MaxPool1d,
                      Linear,
                      Dropout,
                      BCEWithLogitsLoss)

from torch_geometric.transforms import RandomLinkSplit
from torch_geometric.data import Data
from torch_geometric.loader import DataLoader
from torch_geometric.nn import aggr
from torch_geometric.utils import (k_hop_subgraph,
                                   to_scipy_sparse_matrix)
```

2. Загружаем набор данных `Cora`, формируем обучающий, проверочный и тестовый наборы данных, используя случайное разбиение графовых данных на уровне связей:

```
set_seed()

# загружаем набор данных Cora
transform = RandomLinkSplit(num_val=0.05,
                            num_test=0.1,
                            is_undirected=True,
                            split_labels=True)
```

```
dataset = Planetoid('.', name='Cora', transform=transform)
train_data, val_data, test_data = dataset[0]
```

3. Случайное разбиение графовых данных на уровне связей создает новые поля в объекте `Data` для хранения метки и индекса каждого положительного (реального) и отрицательного (фиктивного) ребра:

```
train_data
```

```
Data(x=[2708, 1433], edge_index=[2, 8976], y=[2708], train_mask=[2708], val_
mask=[2708], test_mask=[2708], pos_edge_label=[4488], pos_edge_label_index=[2,
4488], neg_edge_label=[4488], neg_edge_label_index=[2, 4488])
```

4. Пишем функцию для обработки каждого набора и получения охватывающих подграфов с признаками узлов и one-hot-закодированными метками узлов. Мы создаем список для хранения этих подграфов:

```
def seal_processing(dataset, edge_label_index, y):
    data_list = []
```

5. Для каждой пары узлов (узла-отправления и узла-назначения) в наборе данных извлекаем соседей k-го уровня (в данном случае $k = 2$):

```
    for src, dst in edge_label_index.t().tolist():
        sub_nodes, sub_edge_index, mapping, _ = k_hop_subgraph(
            [src, dst], 2, dataset.edge_index, relabel_nodes=True
        )
        src, dst = mapping.tolist()
```

6. Вычисляем расстояния с использованием функции DRNL. Сначала мы выполняем фильтрацию ребер в подграфе, оставляем лишь уникальные ребра, связанные с направлением (src, dst) и (dst, src):

```
        # выполняем фильтрацию ребер в подграфе
        mask1 = (sub_edge_index[0] != src) | (sub_edge_index[1] != dst)
        mask2 = (sub_edge_index[0] != dst) | (sub_edge_index[1] != src)
        sub_edge_index = sub_edge_index[:, mask1 & mask2]
```

7. Вычисляем матрицы смежности для узлов-отправлений и узлов-назначений на основе предыдущего подграфа:

```
        # разметка узлов на основе двойного радиуса (DRNL)
        src, dst = (dst, src) if src > dst else (src, dst)
        adj = to_scipy_sparse_matrix(
            sub_edge_index, num_nodes=sub_nodes.size(0)).tocsr()

        idx = list(range(src)) + list(range(src + 1, adj.shape[0]))
        adj_wo_src = adj[idx, :][:, idx]

        idx = list(range(dst)) + list(range(dst + 1, adj.shape[0]))
        adj_wo_dst = adj[idx, :][:, idx]
```

8. Вычисляем расстояние между каждым узлом и целевым узлом (узлом-отправлением или узлом-назначением):

```
        # вычисляем расстояние между каждым узлом и целевым узлом-источником
        d_src = shortest_path(adj_wo_dst, directed=False,
                              unweighted=True, indices=src)
        d_src = np.insert(d_src, dst, 0, axis=0)
        d_src = torch.from_numpy(d_src)

        # вычисляем расстояние между каждым узлом и целевым узлом-отправлением
        d_dst = shortest_path(adj_wo_src, directed=False,
                              unweighted=True, indices=dst-1)
        d_dst = np.insert(d_dst, src, 0, axis=0)
        d_dst = torch.from_numpy(d_dst)
```

9. Вычисляем метки узлов z для каждого узла в подграфе:

```
        # вычисляем метку z для каждого узла
        dist = d_src + d_dst
        z = 1 + torch.min(d_src, d_dst) + dist // 2 * (dist // 2 + dist % 2 - 1)
        z[src], z[dst], z[torch.isnan(z)] = 1., 1., 0.
        z = z.to(torch.long)
```

10. В этом примере не будем использовать эмбеддинги узлов, но все равно выполним конкатенацию признаков и one-hot-закодированных меток узлов для получения матрицы с информацией об узлах:

```
        # конкатенируем признаки узлов и one-hot-закодированные метки
        # узлов (с фиксированным количеством классов)
        node_labels = F.one_hot(z, num_classes=200).to(torch.float)
        node_emb = dataset.x[sub_nodes]
        node_x = torch.cat([node_emb, node_labels], dim=1)
```

11. Создаем объект `Data` и добавляем его в ранее созданный список, который является итоговым результатом выполнения этой функции:

```
        # создаем объект Data
        data = Data(x=node_x, z=z, edge_index=sub_edge_index, y=y)
        data_list.append(data)

    return data_list
```

12. Давайте воспользуемся нашей функцией для извлечения охватывающих подграфов для каждого набора данных. Мы разделяем положительные и отрицательные примеры, чтобы получить правильную метку для прогноза:

```
# извлечение охватывающих подграфов
train_pos_data_list = seal_processing(
    train_data, train_data.pos_edge_label_index, 1)
train_neg_data_list = seal_processing(
    train_data, train_data.neg_edge_label_index, 0)

val_pos_data_list = seal_processing(
    val_data, val_data.pos_edge_label_index, 1)
val_neg_data_list = seal_processing(
```

```
    val_data, val_data.neg_edge_label_index, 0)

test_pos_data_list = seal_processing(
    test_data, test_data.pos_edge_label_index, 1)
test_neg_data_list = seal_processing(
    test_data, test_data.neg_edge_label_index, 0)
```

13. Затем объединяем списки положительных и отрицательных примеров для воссоздания обучающего, проверочного и тестового наборов данных:

```
train_dataset = train_pos_data_list + train_neg_data_list
val_dataset = val_pos_data_list + val_neg_data_list
test_dataset = test_pos_data_list + test_neg_data_list
```

14. Создаем загрузчики данных для обучения GNN с использованием батчей:

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32)
test_loader = DataLoader(test_dataset, batch_size=32)
```

15. Создаем новый класс для модели DGCNN. Параметр представляет собой количество узлов, которое следует сохранить для каждого подграфа.

```
class DGCNN(torch.nn.Module):
    def __init__(self, dim_in, k=30):
        super().__init__()
```

16. Создаем четыре GCN-слоя (слоя графовой свертки) с фиксированной скрытой размерностью 32:

```
        # GCN-слои
        self.gcn1 = GCNConv(dim_in, 32)
        self.gcn2 = GCNConv(32, 32)
        self.gcn3 = GCNConv(32, 32)
        self.gcn4 = GCNConv(32, 1)
```

17. Создаем экземпляр глобального сортировочного пулинга, который является основой архитектуры DGCNN (Deep Graph Convolutional Neural Network):

```
        # глобальный сортировочный пулинг
        self.global_pool = aggr.SortAggregation(k=k)
```

18. Порядок узлов, предлагаемый глобальным пулингом, позволяет нам использовать традиционные сверточные слои:

```
        # слои свертки
        self.conv1 = Conv1d(1, 16, 97, 97)
        self.conv2 = Conv1d(16, 32, 5, 1)
        self.maxpool = MaxPool1d(2, 2)
```

19. Наконец, для получения прогнозов используем многослойный перцептрон (MLP):

```
        # плотные слои
        self.linear1 = Linear(352, 128)
        self.dropout = Dropout(0.5)
        self.linear2 = Linear(128, 1)
```

20. В функции `forward()` получаем эмбеддинги узлов для каждого GCN-слоя и конкатенируем результаты:

```
    def forward(self, x, edge_index, batch):
        # 1. слои графовой свертки
        h1 = self.gcn1(x, edge_index).tanh()
        h2 = self.gcn2(h1, edge_index).tanh()
        h3 = self.gcn3(h2, edge_index).tanh()
        h4 = self.gcn4(h3, edge_index).tanh()
        h = torch.cat([h1, h2, h3, h4], dim=-1)
```

21. Полученный результат последовательно передаем в слой глобального сортировочного пулинга, сверточные слои и плотные слои:

```
        # 2. глобальный сортировочный пулинг
        h = self.global_pool(h, batch)

        # 3. традиционные слои свертки и плотные слои
        h = h.view(h.size(0), 1, h.size(-1))
        h = self.conv1(h).relu()
        h = self.maxpool(h)
        h = self.conv2(h).relu()
        h = h.view(h.size(0), -1)
        h = self.linear1(h).relu()
        h = self.dropout(h)
        h = self.linear2(h).sigmoid()

        return h
```

22. Если есть возможность, обучаем модель на GPU, используем оптимизатор Adam и бинарную кросс-энтропию в качестве функции потерь:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = DGCNN(train_dataset[0].num_features).to(device)
optimizer = torch.optim.Adam(params=model.parameters(), lr=0.0001)
criterion = BCEWithLogitsLoss()
```

23. Пишем традиционную функцию `train()` для обучения на батчах:

```
set_seed()

def train():
    model.train()
    total_loss = 0
```

```
    for data in train_loader:
        data = data.to(device)
        optimizer.zero_grad()
        out = model(data.x, data.edge_index, data.batch)
        loss = criterion(out.view(-1), data.y.to(torch.float))
        loss.backward()
        optimizer.step()
        total_loss += float(loss) * data.num_graphs

    return total_loss / len(train_dataset)
```

24. В функции `test()` мы вычисляем AUC-ROC и среднюю точность для сравнения качества модели SEAL с качеством модели VGAE:

```
@torch.no_grad()
def test(loader):
    model.eval()
    y_pred, y_true = [], []

    for data in loader:
        data = data.to(device)
        out = model(data.x, data.edge_index, data.batch)
        y_pred.append(out.view(-1).cpu())
        y_true.append(data.y.view(-1).cpu().to(torch.float))

    auc = roc_auc_score(torch.cat(y_true), torch.cat(y_pred))
    ap = average_precision_score(torch.cat(y_true), torch.cat(y_pred))

    return auc, ap
```

25. Обучаем модель DGCNN, задав 31 эпоху:

```
for epoch in range(31):
    loss = train()
    val_auc, val_ap = test(val_loader)
    print(f'Эпоха {epoch:>2}:\n| Функция потерь: {loss:.4f} | '
          f'AUC-ROC на валид. наборе {val_auc:.4f} | '
          f'AP на валид. наборе: {val_ap:.4f}%')
```

```
Эпоха  0:
| Функция потерь: 0.6996 | AUC-ROC на валид. наборе 0.7970 | AP на валид. наборе:
0.8178%
Эпоха  1:
| Функция потерь: 0.6240 | AUC-ROC на валид. наборе 0.8438 | AP на валид. наборе:
0.8706%
.   .   .   .   .
Эпоха 29:
| Функция потерь: 0.5441 | AUC-ROC на валид. наборе 0.8668 | AP на валид. наборе:
0.8757%
Эпоха 30:
| Функция потерь: 0.5438 | AUC-ROC на валид. наборе 0.8664 | AP на валид. наборе:
0.8722%
```

26. Оцениваем качество модели на тестовом наборе:

```
AUC-ROC на тестовом наборе: 0.8487 | AP на тестовом наборе 0.8505
```

Мы получаем результаты, схожие с теми, которые наблюдались при использовании VGAE (AUC-ROC на тестовом наборе – 0.8487 и средняя точность на тестовом наборе – 0.8505). В теории методы, основанные на подграфах, типа SEAL, более выразительны, чем методы, основанные на узлах, типа VGAE. Они захватывают больше информации, явно рассматривая всю окрестность вокруг целевого узла. Кроме того, точность SEAL можно улучшить путем увеличения числа учитываемых соседей с помощью параметра *k*.

Выводы

В этой главе мы исследовали новую задачу – прогнозирование связей. Мы показали, как можно решать эту задачу, представив эвристические и матричные методы. Эвристики можно классифицировать в зависимости от соседей *k*-го уровня, которых они рассматривают, – на локальные, рассматривающие соседей 1-го уровня, и глобальные, полностью исследующие весь граф. Напротив, матричная факторизация аппроксимирует матрицу смежности с помощью эмбеддингов узлов. Мы также пояснили, как этот метод связан с алгоритмами, описанными в предыдущих главах (DeepWalk и Node2Vec).

После этого краткого введения мы узнали, как прогнозировать связи с помощью графовых нейронных сетей (GNN). Мы выделили два вида методов на основе эмбеддингов узлов (GAE и VGAE) и представлениях подграфов (SEAL). Наконец, реализовали VGAE и SEAL на наборе данных `Cora`, используя случайное разбиение графового набора данных на уровне связей и отрицательное семплирование. Обе модели достигли сопоставимого качества, хотя SEAL строго более выразителен. В главе 11 «Генерация графов с помощью графовых нейронных сетей» мы рассмотрим различные стратегии для создания реалистичных графов. Сначала опишем традиционные методы с использованием популярной модели Эрдеша–Реньи. Затем посмотрим, как работают глубокие генеративные методы, заново воспользовавшись GVAE и представив новую архитектуру – **графовую рекуррентную нейронную сеть** (Graph Recurrent Neural Network – GraphRNN).

Дополнительное чтение

[1] H. Tong, C. Faloutsos and J.-y. Pan. Fast Random Walk with Restart and Its Applications in *Sixth International Conference on Data Mining (ICDM'06)*, 2006, pp. 613–622, doi: 10.1109/ICDM.2006.70.

[2] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. *Matrix Factorization Techniques for Recommender Systems*. Computer 42, 8 (August 2009), 30–37. https://doi.org/10.1109/MC.2009.263.

[3] J. Qiu, Y. Dong, H. Ma, J. Li, K. Wang, and J. Tang. *Network Embedding as Matrix Factorization*. Feb. 2018. doi: 10.1145/3159652.3159706.

[4] D. P. Kingma and M. Welling. *Auto-Encoding Variational Bayes*. arXiv, 2013. doi: 10.48550/ARXIV.1312.6114.

[5] T. N. Kipf and M. Welling. *Variational Graph Auto-Encoders*. arXiv, 2016. doi: 10.48550/ARXIV.1611.07308.

[6] M. Zhang and Y. Chen. *Link Prediction Based on Graph Neural Networks*. arXiv, 2018. doi: 10.48550/ARXIV.1802.09691.

[7] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. *An end-to-end deep learning architecture for graph classification*. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence* and *Thirtieth Innovative Applications of Artificial Intelligence Conference* and *Eighth AAAI Symposium on Educational Advances in Artificial Intelligence* (AAAI'18/IAAI'18/EAAI'18). AAAI Press, Article 544, 4438–4445.

Глава 11

Генерация графов с помощью графовых нейронных сетей

Генерация графов включает в себя поиск методов создания новых графов. Речь идет об анализе функционирования и эволюции графа. Она также применяется в аугментации данных, обнаружении аномалий, поиске лекарств и т. д. Мы можем выделить два типа генерации: **реалистичную генерацию графов** (realistic graph generation), имитирующую заданный граф (например, в аугментации данных), и **целенаправленную генерацию графов** (goal-directed graph generation), создающую графы, оптимизирующие конкретную метрику (например, при генерации молекул).

В этой главе мы рассмотрим традиционные методы, чтобы понять, как работает генерация графов. Мы сосредоточимся на двух популярных алгоритмах: моделях **Эрдеша–Реньи** (Erdos-Renyi) и **«малого мира»** (small world). Они обладают интересными свойствами, но у них есть и проблемы, которые оправдывают необходимость генерации графов на основе графовых нейронных сетей (GNN). Во втором разделе мы опишем три семейства решений: модели, основанные на вариационных автоэнкодерах (VAE), авторегрессионные модели, модели, основанные на генеративных состязательных сетях (GAN). Наконец, реализуем фреймворк на основе GAN, используя **обучение с подкреплением** (Reinforcement Learning – RL) для генерации новых химических соединений. Вместо PyTorch Geometric мы будем использовать библиотеку **DeepChem** на основе TensorFlow.

К концу этой главы вы сможете генерировать графы с использованием традиционных методов и методов на основе GNN. У вас будет хороший обзор генерации графов и различных задач, которые вы можете решать в рамках этого направления. Вы узнаете, как реализовать гибридную архитектуру, позволяющую направить генерацию на создание валидных молекул с желательными свойствами.

В этой главе мы рассмотрим следующие основные темы:

- «Генерация графов с помощью традиционных методов»,
- «Генерация графов с помощью графовых нейронных сетей»,
- «Генерация молекул с помощью MolGAN».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter11.

Генерация графов с помощью традиционных методов

Традиционные методы генерации графов изучаются десятилетиями. Именно поэтому их легко понять, и они могут использоваться в качестве базовых моделей в различных задачах. Однако нередко они ограничены с точки зрения типов генерируемых графов. Большинство из них настроены на вывод определенных топологий, поэтому они не могут просто имитировать заданную сеть.

В этом разделе мы представим два классических метода: модель Эрдеша–Реньи и модель «малого мира».

Модель Эрдеша–Реньи

Модель Эрдеша–Реньи является самой простой и популярной моделью случайного графа. Она была представлена венгерскими математиками Паулем Эрдешем и Альфредом Реньи в 1959 году [1] и независимо от них была предложена Эдгаром Гилбертом в том же году [2]. Эта модель имеет два варианта: $G(n, p)$ и $G(n, M)$.

Модель $G(n, p)$ проста: у нас есть n узлов и вероятность p соединения пары узлов. Мы пытаемся случайным образом соединить каждый узел с другим, чтобы получить итоговый граф. Это означает, что существует $\binom{n}{2}$ возможных связей. Еще один способ понять вероятность p – рассмотреть ее как параметр, позволяющий изменить плотность сети.

В библиотеке networkx есть непосредственная реализация модели $G(n, p)$.

1. Давайте импортируем библиотеки networkx и matplotlib:

```
import networkx as nx
import matplotlib.pyplot as plt
```

2. Мы генерируем с помощью функции `nx.erdos_renyi_graph()` граф `G` с 10 узлами (n = 10) и вероятностью появления ребра 0.5 (p = 0.5):

```
G = nx.erdos_renyi_graph(10, 0.9, seed=0)
```

3. Мы укладываем полученные узлы с использованием функции `nx.circular_layout()`. Можно использовать другие укладки, но именно эта укладка удобна для сравнения различных значений параметра p:

```
pos = nx.circular_layout(G)
```

4. Отрисовываем граф `G` с помощью укладки pos, используя функцию `nx.draw_networkx()`. Глобальные эвристики обычно более точны, но требуют информации о всем графе. Однако это не единственный способ прогнозирования связей с использованием подобной информации:

```
nx.draw_networkx(G, pos=pos)
```

Получаем следующий вывод:

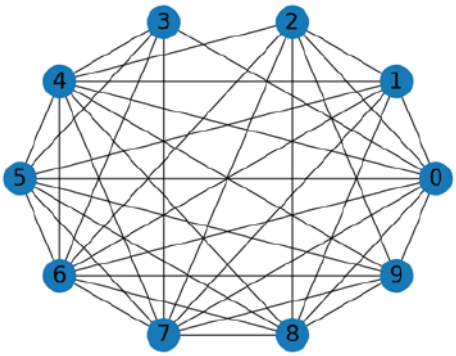


Рис. 11.1 ❖ Граф Эрдеша–Реньи с 10 узлами и p=0.5

Мы можем повторить этот процесс с вероятностями 0.1 и 0.9, чтобы получить результаты, приведенные на рис. 11.2.

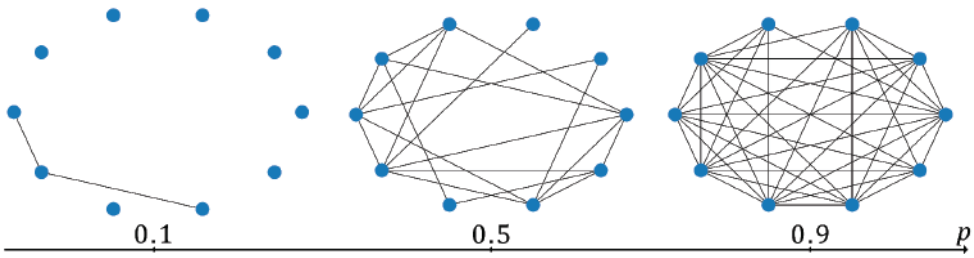


Рис. 11.2 ❖ Граф Эрдеша–Реньи с различными вероятностями появления ребра

Мы видим, что многие узлы изолированы при низкой вероятности p, в то время как при высокой вероятности p узлы в графе сильно взаимосвязаны.

В модели $G(n, M)$ мы случайным образом выбираем граф из всех графов с n узлами и M ребрами. Например, если $n = 3$ и $M = 2$, существует три возможных графа (см. рис. 11.3). Модель $G(n, M)$ просто случайным образом выбирает один из этих графов. Это другой подход к решению той же самой задачи, но он не так популярен, как модель $G(n, p)$, потому что в целом он более сложен для анализа.

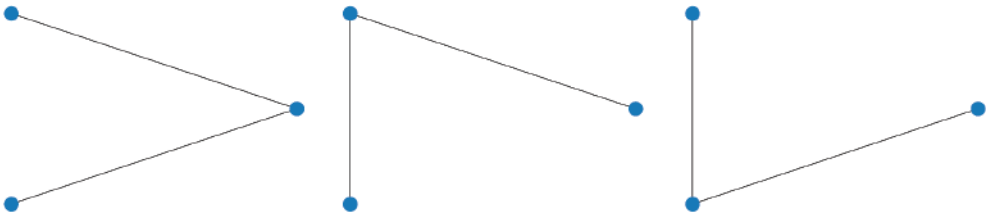


Рис. 11.3 ❖ Набор графов с тремя узлами и двумя ребрами

Мы можем реализовать модель $G(n, M)$ в Python с помощью функции `nx.gnm_random_graph()`:

```
G = nx.gnm_random_graph(3, 2, seed=3)
pos = nx.circular_layout(G)
nx.draw_networkx(G, pos=pos)
```

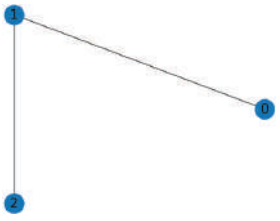


Рис. 11.4 ❖ Граф, случайно выбранный
из набора графов с тремя узлами и двумя ребрами

Самым сильным и наиболее интересным предположением, выдвигаемым моделью $G(n, p)$, является предположение о том, что связи между узлами являются независимыми (т. е. они не взаимодействуют друг с другом). К сожалению, это не соответствует действительности для большинства графов реального мира, в которых мы наблюдаем кластеры и сообщества, противоречащие этому правилу.

Модель «малого мира»

Представленная в 1998 году Дунканом Уоттсом и Стивеном Строгацем [3], модель «малого мира» пытается имитировать поведение биологических, технологических и социальных сетей. Основная идея заключается в том, что сети реального мира не являются полностью случайными (как в модели Эрдеша–Реньи), но и не полностью регулярными[1] (как в решетке). Этот тип топологии находится где-то посередине, поэтому его можно интерполировать с помощью коэффициента. Модель «малого мира» создает графы, которые обладают двумя следующими свойствами:

- **короткими путями** (short paths): среднее расстояние между любыми двумя узлами в сети является относительно небольшим, что облегчает быстрое распространение информации по всей сети;
- **высокими коэффициентами кластеризации** (high clustering coefficients): узлы в сети имеют тенденцию быть тесно связанными друг с другом, образуя плотные кластеры узлов.

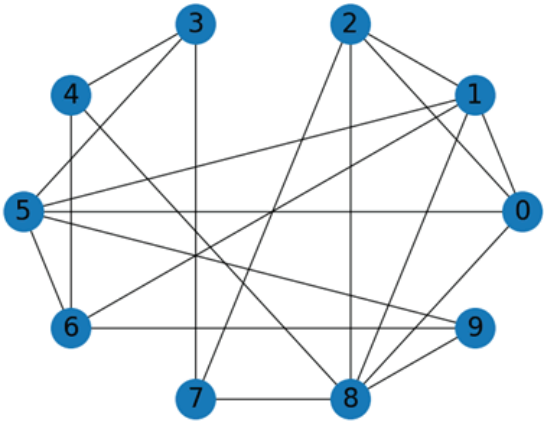


Рис. 11.5 ❖ Сеть «малого мира», полученная с помощью модели Уоттса–Строгаца

Многие алгоритмы обнаруживают свойства «малого мира». В следующем разделе мы опишем оригинальную модель Уоттса–Строгаца, предложенную в [3]. Ее можно реализовать, выполнив следующие шаги.

1. Мы инициализируем граф с n узлами.
2. Каждый узел соединяем с его k ближайшими соседями (или k – 1 соседями, если k является нечетным).
3. У каждой связи между узлами i и j есть вероятность быть замененной на связь между i и другим узлом k, где k – это еще один случайный узел.

[1] *Регулярный* (однородный) *граф* – это *граф*, степени всех вершин которого равны, т. е. каждая вершина имеет одинаковое количество соседей. – *Прим. перев.*

Проще говоря, мы осуществляем переключение (rewiring), в ходе которого исходная связь между *i* и *j* удаляется и заменяется другой связью между *i* и *k*.

В Python мы можем реализовать этот алгоритм, вызвав функцию `nx.watts_strogatz_graph()`:

```
G = nx.watts_strogatz_graph(10, 4, 0.5, seed=0)
pos = nx.circular_layout(G)
nx.draw_networkx(G, pos=pos)
```

Как и в случае с моделью Эрдеша–Реньи, мы можем повторить этот процесс с разными вероятностями переключения *p*.

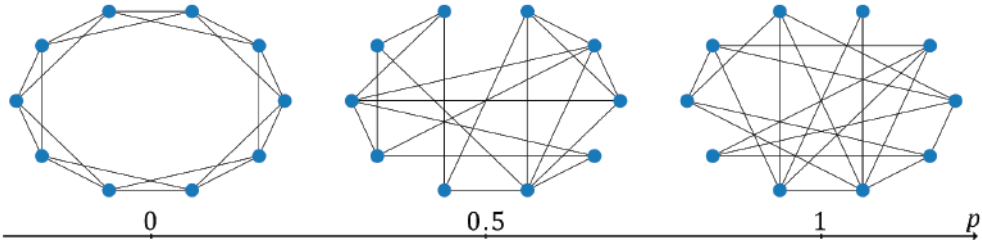


Рис. 11.6 ❖ Модель «малого мира» с разными вероятностями переключения

Мы видим, что при значении $p = 0$ граф полностью регулярен (у каждой вершины одинаковое количество соседей). На противоположном конце при $p = 1$ граф полностью случаен, поскольку каждая связь в нем была переподключена. Между этими двумя крайними случаями находится сбалансированный граф с узлами-хабами и локальной кластеризацией.

Однако модель Уоттса–Строгаца не создает реалистичное распределение степеней. Кроме того, она требует фиксированное количество узлов, это означает, что ее нельзя использовать для роста сети. В общем, классические методы не могут полностью охватить все разнообразие и сложность реальных графов. Это подтолкнуло к созданию нового семейства методов, часто называемых генерацией графов с помощью глубоких нейронных сетей.

Генерация графов с помощью графовых нейронных сетей

Глубокие графовые генеративные модели представляют собой архитектуры на основе графовых нейронных сетей (GNN), которые более выразительны по сравнению с традиционными методами. Однако за этим стоит определенная цена: они, как и классические методы, часто слишком сложны для анализа и понимания. Мы выделяем три основных семейства архитектур

для глубокой генерации графов: вариационные автоэнкодеры (VAE), генеративно-состязательные сети (GAN) и авторегрессионные модели. Существуют и другие методы, такие как нормализующие потоки (normalizing flows) или диффузионные модели (diffusion models), но они являются менее популярными и зрелыми по сравнению с этими тремя.

В данном разделе будет описано, как использовать вариационные автоэнкодеры (VAE), генеративно-состязательные сети (GAN) и авторегрессионные модели для генерации графов.

Вариационные графовые автоэнкодеры

Как видно из последней главы, VAE могут использоваться для аппроксимации матрицы смежности. **Вариационный графовый автоэнкодер** (Variational Graph Autoencoder – VGAE), который мы уже разбирали, состоит из двух компонентов: энкодера (кодировщика) и декодера (декодировщика). Напомним, что кодировщик обычно состоит из трех GCN-слоев: общего слоя и двух слоев для аппроксимации параметров распределения. Цель заключается в том, чтобы выучить параметры каждого скрытого нормального распределения – среднее μ_i (выучиваем с помощью GCN_μ-слоя) и дисперсию σ_i^2 (на практике логарифм стандартного отклонения, выучиваем с помощью GCN_σ-слоя).

Декодировщик семплирует эмбеддинги z_i из выученных распределений с помощью трюка перепараметризации. Затем он использует скалярное произведение скрытых переменных для аппроксимации матрицы смежности $\hat{A} = \sigma(Z^T Z)$.

В предыдущей главе мы использовали $\hat{A}$ для прогнозирования связей. Однако это не единственное ее применение: она непосредственно дает нам матрицу смежности сети, которая имитирует графы, увиденные во время обучения. Вместо прогнозирования связей мы можем использовать этот вывод для создания новых графов. Вот пример матрицы смежности, созданной моделью VGAE из главы 10 (здесь для экономии места приведем только получение матрицы смежности):

```
z = model.encode(test_data.x, test_data.edge_index)
adj = torch.where((z @ z.T) > 0.9, 1, 0)
adj
```

```
tensor([[1, 0, 0,  ..., 0, 1, 1],
        [0, 1, 1,  ..., 0, 0, 0],
        [0, 1, 1,  ..., 0, 1, 1],
        ...,
        [0, 0, 0,  ..., 1, 0, 0],
        [1, 0, 1,  ..., 0, 1, 1],
        [1, 0, 1,  ..., 0, 1, 1]])
```

Начиная с 2016 года, были предприняты попытки расширить этот метод за пределы модели VGAE для вывода признаков узлов и ребер. Хорошим при-

мером является одна из самых популярных графовых генеративных моделей на основе вариационных автоэнкодеров: GraphVAE [4]. Представленная в 2018 году Симоновским и Комодакисом, она разработана для генерации реалистичных молекул. Для этого требуется способность различать узлы (атомы) и ребра (химические связи).

GraphVAE рассматривает графы $G = (A, E, F)$, где A – матрица смежности, E – тензор атрибутов ребер и F – матрица атрибутов узлов. Она выучивает вероятностную модель графа $\tilde{G} = (\tilde{A}, \tilde{E}, \tilde{F})$ с заранее определенным количеством узлов. В этой вероятностной модели $\tilde{A}$ содержит вероятности узлов ($\tilde{A}_{a,a}$) и ребер ($\tilde{A}_{a,b}$), $\tilde{E}$ – вероятности классов для ребер, $\tilde{F}$ содержит вероятности классов для узлов.

По сравнению с GVAE кодировщик GraphVAE представляет собой нейронную сеть прямого распространения с **графовыми свертками, учитывающими информацию о ребрах** (edge-conditional graph convolution – ECC), а его декодировщик – многослойный перцептрон (MLP) с тремя выходами. Вся архитектура кратко представлена на рис. 11.7.

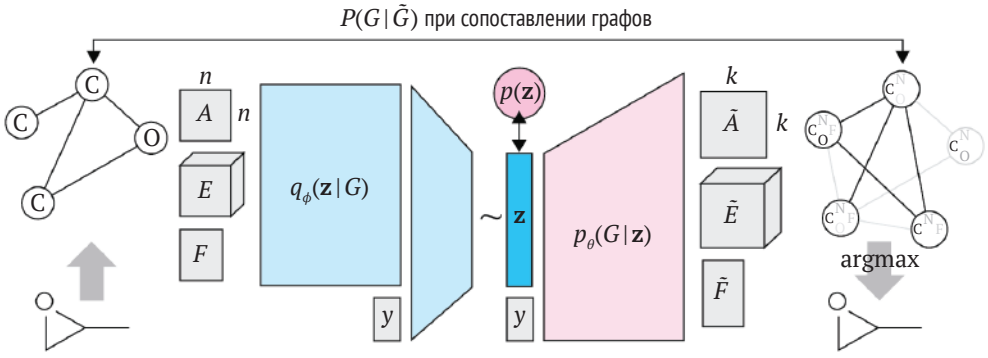


Рис. 11.7 ❖ Процесс инференса GraphVAE

Существует множество других архитектур для генерации графов на основе вариационных автоэнкодеров. Однако их роль не ограничивается имитацией графов, они могут включать в себя ограничения, определяющие тип генерируемых графов.

Популярным способом добавления этих ограничений является их проверка во время фазы декодирования, например в **вариационном графовом автоэнкодере с ограничениями** (Constrained Variational Graph Autoencoder – CVGAE) [5]. В данной архитектуре кодировщик представляет собой **графовую сверточную сеть с управляемым доступом**, или **управляемую графовую сверточную сеть** (Gated Graph Convolutional Network – GGCN), а декодировщик – авторегрессионную модель. Авторегрессионные декодировщики особенно подходят для этой задачи, поскольку они могут проверять каждое ограничение на каждом этапе процесса. Наконец, еще одной техникой добавления ограничений является использование регуляризаторов на основе

лагранжиана, которые вычисляются быстрее, но позволяют больше вариаций («менее строги») в плане генерации графов [6].

Авторегрессионные модели

Кроме того, авторегрессионные могут использоваться самостоятельно. Отличие от других моделей заключается в том, что предыдущие выходы становятся частью текущего входа. В этой структуре генерация графа становится последовательным процессом принятия решений, который учитывает как данные, так и предыдущие решения. Например, на каждом этапе авторегрессионная модель может создавать новый узел или новую связь. Затем полученный граф подается на вход модели для следующего этапа генерации, пока мы не прекратим этот процесс. Схема этого процесса приведена на рис. 11.8.

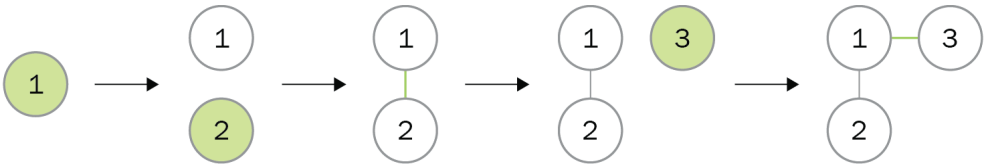


Рис. 11.8 ❖ Авторегрессионный процесс генерации графов

На практике мы используем рекуррентные нейронные сети (Recurrent Neural Networks – RNN) для реализации этой авторегрессионной схемы. В данной архитектуре предыдущие выходы используются в качестве входов для вычисления текущего скрытого состояния. Кроме того, они способны обрабатывать входы произвольной длины, что критически важно для итеративной генерации графов. Однако такая обработка требует больше времени по сравнению с сетями прямого распространения, поскольку для получения окончательного вывода необходимо обработать всю последовательность. Два наиболее популярных типа RNN – это **управляемые реккурентные блоки** (Gated Recurrent Units – GRU) и **нейронная сеть с долговременной краткосрочной памятью** (Long Short-Term Memory – LSTM).

GraphRNN [7], предложенная в 2018 году Цзясюанем Ю совместно с коллегами, – это прямая реализация вышеизложенных техник для глубокой генерации графов. В данной архитектуре используются две RNN:

- *RNN, работающая на уровне графа*, которая генерирует последовательность узлов (включая начальное состояние);
- *RNN, работающая на уровне ребер*, которая прогнозирует связи для каждого вновь добавленного узла.

RNN, работающая на уровне ребер, принимает скрытое состояние RNN, работающей на уровне графа, в качестве входа. Проще говоря, информация, полученная на уровне графа, передается как входной сигнал на уровень ребер. Скрытое состояние содержит контекст и информацию о генерации узлов

на уровне графа. После того как RNN, работающая на уровне ребер, приняла скрытое состояние, она использует его в процессе генерации своего собственного вывода. Этот механизм иллюстрируется на рис. 11.9.

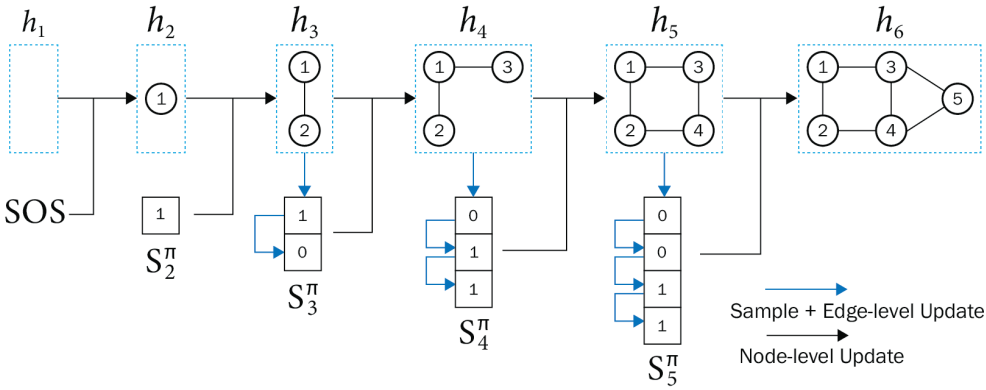


Рис. 11.9 ❖ Архитектура GraphRNN в момент инференса

Обе RNN фактически заполняют матрицу смежности: каждый новый узел, созданный RNN, работающей на уровне графа, добавляет строку и столбец, которые RNN, работающая на уровне ребер, заполняет нулями и единицами. В итоге GraphRNN выполняет следующие шаги.

1. *Добавление нового узла*: RNN, работающая на уровне графа, инициализирует граф, и его выходные данные подаются на вход RNN, работающей на уровне ребер.
2. *Добавление новых связей*: RNN, работающая на уровне ребер, предсказывает, связан ли новый узел с каждым из предыдущих узлов.
3. *Завершение генерации графа*: первые два шага повторяются до тех пор, пока RNN, работающая на уровне ребер, не выдаст токен EOS, обозначающий конец процесса.

GraphRNN может выучивать различные типы графов (сетки, социальные сети, белки и т. д.) и полностью превосходить традиционные техники. Это предпочтительная архитектура для имитации заданных графов, которая обычно дает лучшее качество, чем GraphVAE.

Генеративные состязательные сети (GAN)

Как и VAE, GAN является широко известной генеративной моделью, применяемой в машинном обучении. В рамках этой архитектуры две нейронные сети соревнуются в игре с нулевой суммой, преследуя две разные цели. Первая нейронная сеть – генератор, который создает новые данные, а вторая – дискриминатор, который классифицирует каждый пример как реальный (из обучающего набора) или поддельный (созданный генератором).

За эти годы было предложено два основных улучшения оригинальной архитектуры. Первое из них называется **генеративные состязательные сети Вассерштейна** (Wasserstein GAN – WGAN). В них улучшена стабильность обучения путем минимизации расстояния Вассерштейна между двумя вероятностными распределениями. Эта вариация дополнительно усовершенствована введением штрафа, зависящего от нормы градиента (градиентного штрафа), вместо исходной схемы клиппинга градиентов.

В нескольких работах применили этот подход к глубокой генерации графов. Как и предыдущие методы, генеративные состязательные сети (GAN) могут имитировать графы или создавать сети, оптимизируя определенные ограничения. Последний вариант удобен в таких задачах, как поиск новых химических соединений с определенными свойствами. Эта проблема чрезвычайно обширна (более 10^{60} возможных комбинаций) и сложна из-за своей дискретной природы.

Предложенная Де Као и Кипфом в 2018 году [8] **молекулярная генеративная состязательная сеть GAN** (MolGAN) является популярным решением этой проблемы. Это решение включает WGAN с градиентным штрафом, который стабилизирует обучение при работе с графовыми структурами данных, и целевую функцию на основе обучения с подкреплением (RL-оптимизацию) для создания молекул с желаемыми химическими свойствами. RL-оптимизация основана на алгоритме градиентов глубокой детерминированной политики[1] (Deep Deterministic Policy Gradient – DDPG), модель «актор–критик»[2] вне политики и использующей градиенты детерминированной политики. Архитектура MolGAN кратко представлена на рис. 11.10.

Эта структура разделена на три основных компонента:

- **генератор** (generator) представляет собой многослойный перцептрон (MLP), который выводит матрицу узлов X, содержащую типы атомов, и матрицу смежности A, которая фактически является тензором, содержащим как ребра, так и типы связей. Генератор обучается с использованием линейной комбинации WGAN и функции потерь RL. Мы преобразовываем эти плотные представления в разреженные объекты ($\tilde{X}$ и $\tilde{A}$) с помощью категориального семплирования;

---

[1] Политика глубокого обучения с подкреплением относится к одной из двух категорий: стохастической или детерминированной. Детерминированная политика – это политика, в которой состояния сопоставляются с действиями, что означает, что, когда политике предоставляется информация о состоянии, возвращается действие. Между тем стохастические политики возвращают распределение вероятностей для действий вместо одного дискретного действия. Детерминированные политики используются, когда нет неопределенности в отношении результатов действий, которые могут быть предприняты. Другими словами, когда сама среда детерминирована. Напротив, стохастические результаты политики подходят для условий, в которых результат действий не определен. Обычно сценарии обучения с подкреплением предполагают некоторую степень неопределенности, поэтому используются стохастические политики. – *Прим. перев.*

[2] Актор отвечает за выбор действий, а критик оценивает эффективность или «ценность» этих действий. – *Прим. перев.*

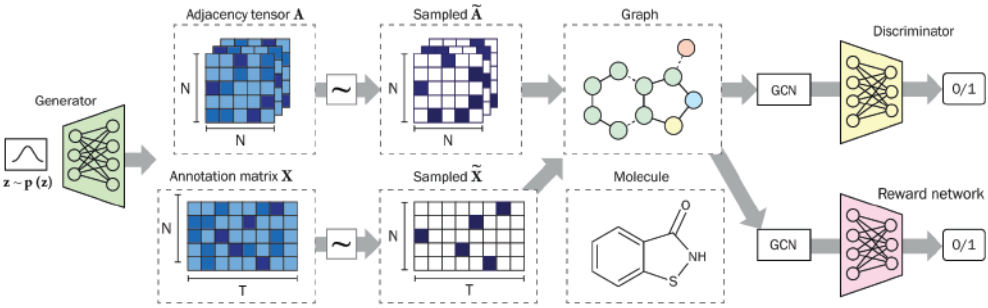


Рис. 11.10 ❖ Архитектура MolGAN в момент инференса

- **дискриминатор** (discriminator) получает графы от генератора и из набора данных и учится их различать. Его обучают исключительно с помощью функции потерь WGAN;
- **сеть вознаграждений** (reward network) присваивает оценку каждому графу. Она обучается с помощью функции потерь MSE на основе реальной оценки, полученной от внешней системы (в данном случае RDKit). Дискриминатор и сеть вознаграждений используют режим GNN, а именно Relational-GCN, вариант GCN, поддерживающий несколько типов ребер. После нескольких слоев графовых сверток эмбеддинги узлов агрегируются в выходной вектор на уровне графа:

$$h_G = \tanh\left(\sum_{i \in V} \sigma(\text{MLP}_1(h_i, x_i)) \odot \tanh(\text{MLP}_2(h_i, x_i))\right).$$

Здесь σ обозначает логистический сигмоид, MLP_1 и MLP_2 – это два многослойных перцептрона (MLP) с линейным выходом, а $\odot$ – поэлементное умножение. Третий многослойный перцептрон (MLP) дополнительно обрабатывает эмбеддинг графа, чтобы получить значение в диапазоне между 0 и 1 для сети вознаграждений и значение в диапазоне между $-\infty$ и $+\infty$ для дискриминатора.

MolGAN создает валидные химические соединения, оптимизируя такие свойства, как вероятность использования в медицинских задачах, синтезируемость и растворимость. В следующем разделе мы реализуем эту архитектуру для генерации новых молекул.

Генерация молекул с помощью MolGAN

Глубокая генерация графов не имеет хороших реализаций в PyTorch Geometric. Ее основное применение – открытие лекарств, поэтому генеративные модели часто реализуются в специализированных библиотеках. Сейчас существуют две популярные библиотеки на языке Python для машинного обучения в области поиска лекарств: `DeepChem` и `torchdrug`. В этом разделе мы будем использовать DeepChem, так как она более зрелая и напрямую реализует MolGAN.

Давайте посмотрим, как мы можем моделировать молекулы с помощью DeepChem и tensorflow. Процедура основана на примере из руководства по DeepChem.

1. Устанавливаем DeepChem (https://deepchem.io), для нее потребуются следующие библиотеки: tensorflow, joblib, NumPy, pandas, scikit-learn, SciPy и rdkit:

```
!pip install deepchem==2.7.1
```

2. Далее мы импортируем необходимые библиотеки, классы и функции:

```
import pandas as pd
import numpy as np
from tensorflow import one_hot

import deepchem as dc
from deepchem.models.optimizers import ExponentialDecay
from deepchem.models import BasicMolGANModel as MolGAN
from deepchem.feat.molecule_featurizers.molgan_featurizer import GraphMatrix

from rdkit import Chem
from rdkit.Chem import Draw
from rdkit.Chem import rdmolfiles
from rdkit.Chem import rdmolops
from rdkit.Chem.Draw import IPythonConsole
```

3. Загружаем набор данных `tox21` (*Toxicology in the 21st Century*), который включает в себя более 6000 химических соединений для анализа их токсичности. В данном примере нам нужны только их упрощенные молекулярные представления в формате строк (**система упрощенного представления молекул в строке ввода** – Simplified Molecular-Input Line-Entry System, или SMILES):

```
_, datasets, _ = dc.molnet.load_tox21()
df = pd.DataFrame(datasets[0].ids, columns=['smiles'])
df
```

	smiles
0	CC(O)(P(=O)(O)O)P(=O)(O)O
1	CC(C)(C)OOC(C)(C)CCC(C)(C)OOC(C)(C)C
2	OC[C@H](O)[C@@H](O)[C@H](O)CO
3	CCCCCCCC(=O)[O-].CCCCCCCC(=O)[O-].[Zn+2]
4	CC(C)COC(=O)C(C)C
...	...
6259	CC1CCCCN1CCCOC(=O)c1ccc(OC2CCCCC2)cc1
6260	Cc1cc(CCCOc2c(C)cc(-c3noc(C(F)(F)F)n3)cc2C)on1
6261	O=C1OC(OC(=O)c2cccnc2Nc2cccc(C(F)(F)F)c2)c2ccc...
6262	CC(=O)C1(C)CC2=C(CCCC2(C)C)CC1C
6263	CC(C)CCC[C@@H](C)[C@H]1CC(=O)C2=C3CC[C@H]4C[C@...

6264 rows × 1 columns

4. Мы рассматриваем только молекулы с максимальным количеством 15 атомов. Отфильтровываем наш набор данных и создаем конструктор признаков (`featurizer`), чтобы преобразовать строковые представления SMILES во входные признаки:

```
# определяем максимальное количество атомов
max_atom = 15

# создаем экземпляр класса MolGanFeaturizer,
# по сути, создаем конструктор признаков
featurizer = dc.feat.MolGanFeaturizer(max_atom_count=max_atom)
# отбираем молекулы с количеством атомов меньше максимального
molecules = [x for x in df['smiles'].values
             if Chem.MolFromSmiles(x).GetNumAtoms() < max_atom]
```

5. Мы вручную пробегаем по нашему набору данных – молекулам – и извлекаем признаки с помощью `featurizer`. Если говорить более подробно, в цикле мы каждый раз создаем объект молекулы на основе строкового представления SMILES и потом получаем канонический порядок атомов в молекуле. Канонический порядок обеспечивает уникальное представление молекулы, не зависящее от начального порядка атомов, что важно, например, для сравнения молекул. Потом перенумеровываем атомы в соответствии с полученным порядком и наконец извлекаем признаки:

```
# создаем список для хранения признаков
features = []

# проход по молекулам и их преобразование в признаки
for x in molecules:
    # создаем объект молекулы из строкового
    # представления SMILES
    mol = Chem.MolFromSmiles(x)
    # получение канонического порядка атомов
    new_order = rdmolfiles.CanonicalRankAtoms(mol)
    # перенумерация атомов в соответствии
    # с каноническим порядком
    mol = rdmolops.RenumberAtoms(mol, new_order)
    # извлечение признаков с помощью featurizer
    feature = featurizer.featurize(mol)
    # проверка наличия признаков перед
    # добавлением в словарь
    if feature.size != 0:
        features.append(feature[0])
```

6. Удаляем невалидные молекулы:

```
# удаляем невалидные молекулы
features = [x for x in features if type(x) is GraphMatrix]
```

7. Затем мы создаем модель `MolGAN`. Для обучения используем темп обучения с экспоненциальным затуханием:

```
# создаем MolGAN
gan = MolGAN(
    learning_rate=ExponentialDecay(0.001, 0.9, 5000),
    vertices=max_atom
)
```

8. Создаем набор данных, чтобы передать его в формате DeepChem в `MolGAN`:

```
# создаем набор данных
dataset = dc.data.NumpyDataset(
    X=[x.adjacency_matrix for x in features],
    y=[x.node_features for x in features]
)
dataset
```

```
<NumpyDataset X.shape: (2107, 15, 15), y.shape: (2107, 15), w.shape: (2107, 1),
task_names: [ 0  1  2 ... 12 13 14]>
```

9. MolGAN использует обучение на основе батчей, поэтому нам необходимо определить итерируемый объект следующим образом:

```
def iterbatches(epochs):
    # итерируем по эпохам
    for i in range(epochs):
        # итерируем по батчам в наборе данных
        for batch in dataset.iterbatches(batch_size=gan.batch_size,
                                         pad_batches=True):
            # преобразовываем матрицы смежности
            # и узлы в one-hot-тензоры
            adjacency_tensor = one_hot(batch[0], gan.edges)
            node_tensor = one_hot(batch[1], gan.nodes)
            # возвращаем словарь с данными для MolGAN
            yield {gan.data_inputs[0]: adjacency_tensor,
                   gan.data_inputs[1]: node_tensor}
```

10. Обучаем модель, задав 25 эпох:

```
# обучаем модель
gan.fit_gan(iterbatches(25),
            generator_steps=0.2,
            checkpoint_interval=5000)
```

11. Мы генерируем 1000 молекул:

```
# генерируем 1000 молекул
generated_data = gan.predict_gan_generator(1000)
generated_mols = featurizer.defeaturize(generated_data)
```

12. Затем проверяем валидность молекул:

```
# проверяем валидность молекулы (стабильность не гарантирована,
# поэтому может оказаться, что количество допустимых
# молекул равно 0
```

```
valid_mols = [x for x in generated_mols if x is not None]
print (f"{len(valid_mols)} валидных молекул "
       f"(из {len((generated_mols))} сгенерированных молекул)")
```

```
807 валидных молекул (из 1000 сгенерированных молекул)
```

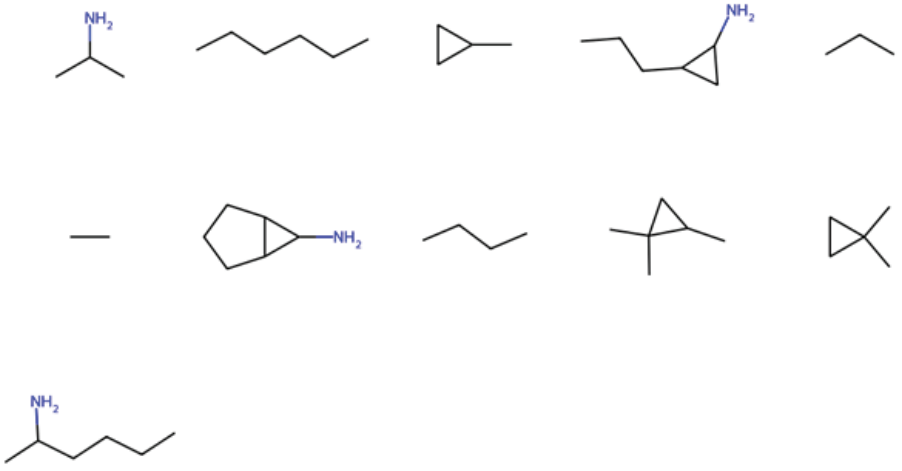
13. Теперь посмотрим, сколько уникальных молекул мы получили:

```
# преобразовываем сгенерированные молекулы
# в строковое представление SMILES
generated_smiles = [Chem.MolToSmiles(x)
                    for x in valid_mols]
# преобразовываем уникальные строковые предствления
# SMILES в молекулы для визуализации
generated_smiles_viz = [Chem.MolFromSmiles(x)
                        for x in set(generated_smiles)]
# печатаем информацию о количестве
# уникальных и избыточных молекул
print(f"{len(generated_smiles_viz)} уникальных валидных молекул "
      f"({len(generated_smiles)-len(generated_smiles_viz)} избыточных молекул)")
```

```
11 уникальных валидных молекул (796 избыточных молекул)
```

14. Наконец, визуализируем сгенерированные молекулы:

```
# визуализируем сгенерированные молекулы
# в виде сетки изображений
Draw.MolsToGridImage(generated_smiles_viz,
                     molsPerRow=5,
                     subImgSize=(200, 200),
                     returnPNG=False)
```



Несмотря на улучшения, достигнутые с помощью GAN, этот процесс обучения довольно нестабилен и может не привести к получению содержательных результатов. Представленный нами код чувствителен к изменениям гипер-

параметров и плохо обобщается на другие наборы данных, включая набор данных `QM9`, использованный в оригинальной статье.

Выводы

В этой главе мы рассмотрели различные методы генерации графов. Сначала мы исследовали традиционные методы на основе вероятностей с интересными математическими свойствами. Однако из-за их ограниченной выразительности мы перешли к методам на основе графовых нейронных сетей (GNN), которые являются гораздо более гибкими. Мы рассмотрели три семейства глубоких генеративных моделей: основанные на вариационных автокодировщиках (VAE), авторегрессионные и основанные на генеративных состязательных сетях (GAN). Мы представили модель из каждого семейства, чтобы понять, как они работают в реальной жизни.

В заключение мы реализовали модель на основе GAN, которая объединяет генератор, дискриминатор и сеть вознаграждений из обучения с подкреплением. Вместо простого имитирования графов, увиденных во время обучения, эта архитектура также способна оптимизировать желаемые свойства типа растворимости. Мы использовали DeepChem и TensorFlow для создания уникальных и валидных молекул. В настоящее время такой подход является общепринятым в индустрии поиска лекарств, где машинное обучение может существенно ускорить разработку лекарств.

В главе 12 «Обучение на гетерогенных графах» мы исследуем новый тип графов, с которыми ранее сталкивались в рекомендательных системах и моделировании молекул. Эти гетерогенные графы содержат несколько типов узлов и/или связей, что требует специальной обработки. Они более универсальные, чем обычные графы, о которых мы говорили, и особенно полезны в таких областях, как графы знаний.

Дополнительное чтение

[1] P. Erdős and A. Renyi. *On random graphs I*, Publicationes Mathematicae Debrecen, vol. 6, p. 290, 1959. Доступ по ссылке https://snap.stanford.edu/class/cs224w-readings/erdos59random.pdf.

[2] E. N. Gilbert, *Random Graphs*, The Annals of Mathematical Statistics, vol. 30, no. 4, pp. 1141–1144, 1959, DOI: 10.1214/aoms/1177706098. Доступ по ссылке https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-30/issue-4/Random-Graphs/10.1214/aoms/1177706098.full.

[3] Duncan J. Watts and Steven H. Strogatz. *Collective dynamics of small-world networks*, Nature, 393, pp. 440–442, 1998. Доступ по ссылке http://snap.stanford.edu/class/cs224w-readings/watts98smallworld.pdf.

[4] M. Simonovsky and N. Komodakis. *GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders* CoRR, vol. abs/1802.03480, 2018 [Online]. Доступ по ссылке http://arxiv.org/abs/1802.03480.

[5] Q. Liu, M. Allamanis, M. Brockschmidt, and A. L. Gaunt. *Constrained Graph Variational Autoencoders for Molecule Design*. arXiv, 2018. DOI: 10.48550/ARXIV.1805.09076. Доступ по ссылке https://arxiv.org/abs/1805.09076.

[6] T. Ma, J. Chen, and C. Xiao, Constrained Generation of Semantically Valid Graphs via Regularizing Variational Autoencoders. arXiv, 2018. DOI: 10.48550/ARXIV.1809.02630. Доступ по ссылке https://arxiv.org/abs/1809.02630.

[7] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec. *GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models*. arXiv, 2018. DOI: 10.48550/ARXIV.1802.08773. Доступ по ссылке https://arxiv.org/abs/1802.08773.

[8] N. De Cao and T. Kipf. *MolGAN: An implicit generative model for small molecular graphs*. arXiv, 2018. DOI: 10.48550/ARXIV.1805.11973. Доступ по ссылке https://arxiv.org/abs/1805.11973.

Глава 12

Обучение на гетерогенных графах

В предыдущей главе мы пытались создать реалистичные молекулы, содержащие различные типы узлов (атомов) и ребра (связи). Кроме того, мы наблюдаем подобное поведение и в других областях, таких как рекомендательные системы (пользователи и товары), социальные сети (подписчики и подписки) или кибербезопасность (маршрутизаторы и серверы). Мы называем такие графы **гетерогенными** (heterogeneous) в отличие от гомогенных графов, которые включают только один тип узла и один тип ребра.

В этой главе мы сделаем обзор всего, что знаем о гомогенных графовых нейронных сетях (GNN). Мы познакомимся с нейронной сетью передачи сообщений для обобщения архитектур, которые успели рассмотреть до этого момента. Этот обзор позволит нам понять, как расширить эту нейронную сеть на гетерогенные сети. Начнем с создания собственного гетерогенного набора данных. Затем преобразуем гомогенные архитектуры в гетерогенные.

В последнем разделе мы рассмотрим другой подход и обсудим архитектуру, специально разработанную для обработки гетерогенных сетей. Мы опишем, как она работает, чтобы лучше понять разницу между этой архитектурой и классической графовой нейронной сетью с механизмом внимания (Graph Attention Network – GAT). Наконец, мы реализуем ее в PyTorch Geometric и сравним результаты с предыдущими методами.

К концу этой главы у вас будет четкое понимание различий между гомогенными и гетерогенными графами. Вы сможете создавать свои собственные гетерогенные наборы данных и преобразовывать традиционные модели для работы с гетерогенными графами. Кроме того, вы сможете реализовать архитектуры, специально разработанные для достижения максимальной эффективности при работе с гетерогенными сетями.

В этой главе мы рассмотрим следующие основные темы:

- «Нейронная сеть передачи сообщений»,
- «Знакомство с гетерогенными графами»,
- «Преобразование гомогенных GNN в гетерогенные GNN»,
- «Реализация иерархической нейронной сети с самовниманием».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter12.

Нейронная сеть передачи сообщений

Прежде чем перейти к исследованию гетерогенных графов, давайте кратко вспомним то, что мы узнали о гомогенных графовых нейронных сетях (GNN). В предыдущих главах мы рассмотрели различные функции для агрегации и объединения признаков из различных узлов. Как мы уже узнали из главы 5, самый простой GNN-слой состоит из суммирования линейных комбинаций признаков соседних узлов (включая сам целевой узел) с использованием матрицы весов. Затем выход предыдущей суммы заменяет предыдущий эмбеддинг целевого узла.

Оператор на уровне узла может быть записан следующим образом:

$$h_i' = \sum_{j \in \mathcal{N}_i} h_j W^T,$$

где $\mathcal{N}_i$ – множество соседних узлов i-го узла (включая сам узел), h_i – эмбеддинг i-го узла, а W^T – матрица весов.

GCN- и GAT-слои добавляли фиксированные и динамические веса к признакам узлов, но сохраняли ту же самую идею. Даже LSTM-оператор в GraphSAGE или агрегатор на основе максимального значения в GIN не меняли основную концепцию GNN-слоя. Если мы посмотрим на все эти варианты, мы можем обобщить GNN-слои в общий фреймворк под названием **нейронная сеть передачи сообщений** (Message Passing Neural Network – MPNN или MP-GNN). Разработанный в 2017 году Гилмером и др. [1], этот фреймворк включает три основные операции:

- **message** (сообщение): каждый узел использует функцию, создающую сообщение для каждого соседа. Сообщение узла может просто состоять из его собственных признаков (как в предыдущем примере) или дополнительно учитывать признаки соседнего узла и признаки ребра;
- **aggregate** (агрегация): каждый узел агрегирует сообщения от своих соседей, используя функцию перестановочной эквивариантности типа суммы в предыдущем примере;
- **update** (обновление): каждый узел обновляет свое собственное представление, используя функцию для объединения своих текущих признаков и агрегированных сообщений.

В операциях message и update могут использоваться самопетли (self-loops). Напомним, самопетли представляют собой ребра, которые соединяют узел с самим собой в графе. Это позволяет узлу учитывать влияние собственных

характеристик при генерации сообщения и свое собственное состояние при обновлении информации.

Эти шаги можно подытожить в одном уравнении:

$$h_i' = \gamma(h_i, \bigoplus_{j \in \mathcal{N}_i} \phi(h_i, h_j, e_{j,i})).$$

Здесь h_i – это узловой эмбеддинг для i-го узла, $e_{j,i}$ – реберный эмбеддинг для связи $j \times i$, ϕ – функция сообщения, $\bigoplus$ – функция агрегации и γ – это функция обновления. На рис. 12.1 приведена иллюстрация MPNN.

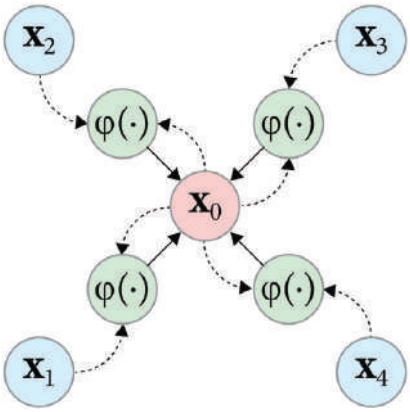


Рис. 12.1 ❖ MPNN

В PyTorch Geometric нейронная сеть передачи сообщений непосредственно реализована в классе `MessagePassing`. Например, вот как можно реализовать GCN-слой с помощью этого класса.

1. Сначала импортируем необходимые библиотеки:

```
from torch.nn import Linear
from torch_geometric.nn import MessagePassing
from torch_geometric.utils import add_self_loops, degree
```

2. Мы объявляем класс `GCNConv`, который наследуется от `MessagePassing`:

```
class GCNConv(MessagePassing):
```

3. Он принимает два параметра – входную размерность и выходную (скрытую) размерность. Инициализируем `MessagePassing` с помощью агрегации `add`. Мы задаем один линейный слой PyTorch без смещения:

```
    def __init__(self, dim_in, dim_h):
        super().__init__(aggr='add')
        self.linear = Linear(dim_in, dim_h, bias=False)
```

4. В функции `forward()` мы реализуем следующую логику. Сначала добавляем самопетли в матрицу смежности, чтобы учитывать целевые узлы:

```
    def forward(self, x, edge_index):
        edge_index, _ = add_self_loops(edge_index,
                                       num_nodes=x.size(0))
```

5. Затем применяем линейное преобразование с помощью ранее заданного линейного слоя:

```
x = self.linear(x)
```

6. Мы вычисляем коэффициент нормализации – $\frac{1}{\sqrt{\deg(i)}\sqrt{\deg(j)}}$:

```
row, col = edge_index
deg = degree(col, x.size(0), dtype=x.dtype)
deg_inv_sqrt = deg.pow(-0.5)
deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]
```

7. Вызываем метод `.propagate()`, передав обновленный `edge_index` (включая самопетли) и наши факторы нормализации, хранящиеся в тензоре `norm`. Внутри этот метод вызывает функции `message()`, `aggregate()` и `update()`. Нам не нужно переопределять `update()`, потому что мы уже включили самопетли. Функция `aggregate()` уже задана на шаге 3 с помощью параметра `aggr='add'`:

```
out = self.propagate(edge_index, x=x, norm=norm)
return out
```

8. Мы переопределяем функцию `message()` для нормализации признаков соседних узлов `x` с помощью фактора нормализации `norm`:

```
    def message(self, x, norm):
        return norm.view(-1, 1) * x
```

9. Теперь создаем экземпляр класса `GCNConv`:

```
conv = GCNConv(16, 32)
```

Этот объект мы будем использовать в качестве GCN-слоя.

Данный пример показывает, как можно создать свои собственные GNN-слои в PyTorch Geometric. Кроме того, вы можете ознакомиться с реализацией GCN- и GAT-слоев в исходном коде. MPNN – важный метод, который поможет нам преобразовать наши графовые нейронные сети в гетерогенные модели.

Знакомство с гетерогенными графами

Гетерогенные графы представляют собой мощный инструмент для представления общих зависимостей между различными сущностями. Наличие различных типов узлов и ребер создает более сложные структуры графов,

но также делает их более сложными для обучения. В частности, одна из основных проблем гетерогенных сетей заключается в том, что признаки различных типов узлов или ребер не обязательно имеют одинаковый смысл или размерность.

Поэтому объединение различных признаков может привести к потере большого объема информации. Однако это неактуально для гомогенных графов, в которых каждая размерность имеет точно такой же смысл для каждого узла или ребра.

Гетерогенные графы – это более общий тип сетей, который может представлять разные типы узлов и ребер. Формально его можно определить как граф $G = (V, E)$, включающий в себя V – множество узлов и E – множество ребер. В гетерогенной среде он связан с отображающей функцией для типов узлов $\phi: V \rightarrow A$ (где A обозначает множество типов узлов) и отображающей функцией для типов связей $\psi: E \rightarrow R$ (где R обозначает множество типов ребер). Первая функция устанавливает соответствие между типами узлов в гетерогенном графе. Вторая функция устанавливает соответствие между типами ребер в гетерогенном графе.

Ниже представлен пример гетерогенного графа.

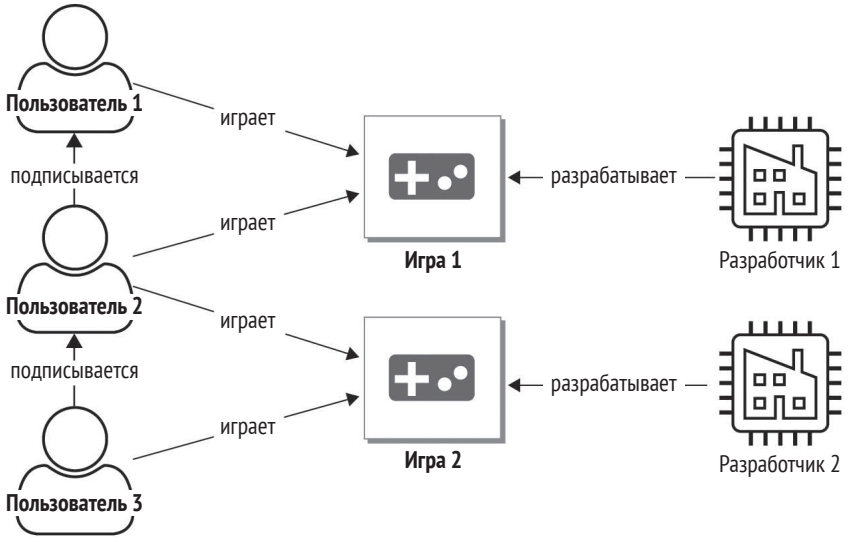


Рис. 12.2 ❖ Пример гетерогенного графа с тремя типами узлов и тремя типами ребер

В этом графе мы видим три типа узлов (пользователи, игры и разработчики) и три типа ребер (**подписывается**, **играет** и **разрабатывает**). Этот граф представляет собой сеть, включающую в себя людей (пользователей и разработчиков) и игры, которая может быть использована для различных приложений, например для рекомендации игр. Если в этом графе содержат-

ся миллионы элементов, его можно использовать в качестве графовой базы знаний или графа знаний. Графы используются Google или Bing для ответа на запросы типа «Кто играет в игры, разработанные **Разработчиком 1**?».

С помощью аналогичных запросов можно извлекать полезные однородные графы. Например, нам нужно рассмотреть только пользователей, играющих в **Игру 1**. Результатом будут **Пользователь 1** и **Пользователь 2**. Мы можем создавать более сложные запросы типа «Кто из пользователей играет в игры, разработанные **Разработчиком 1**?». Результат будет тот же самый, но мы совершили обход двух зависимостей, чтобы получить наших пользователей. Такие запросы называются метапутем.

В первом примере наш метапуть был *Пользователь → Игра → Пользователь* (обычно обозначается как **ПИП**), а во втором – *Пользователь → Игра → Разработчик → Игра → Пользователь* (или **ПИРИП**). Обратите внимание, что тип начального узла и тип конечного узла одинаковы. Метапути – важное понятие в гетерогенных графах, часто используемое для измерения сходства различных узлов.

Теперь давайте посмотрим, как реализовать предыдущий граф с помощью PyTorch Geometric. Мы воспользуемся специальным объектом данных `HeteroData`. Следующие шаги создают объект `data` для хранения графа с рис. 12.2.

1. Мы импортируем класс `HeteroData` из библиотеки `torch_geometric.data` и создаем переменную `data`:

```
from torch_geometric.data import HeteroData
data = HeteroData()
```

2. Сначала давайте сохраним признаки узлов. Мы можем получить доступ к признакам пользователя, например, с помощью `data['user'].x`. Мы передаем тензор размером `[num_users, num_features_users]`. Содержание не имеет значения в этом примере, поэтому мы создадим векторы признаков, заполненные единицами для пользователя 1, двойками для пользователя 2 и тройками для пользователя 3:

```
# [num_users, num_features_users]
data['user'].x = torch.Tensor([[1, 1, 1, 1],
                               [2, 2, 2, 2],
                               [3, 3, 3, 3]])
```

3. Мы повторяем этот процесс с играми и разработчиками. Обратите внимание, что размерность векторов признаков не одинакова. Это важное преимущество гетерогенных графов при работе с разными представлениями:

```
data['game'].x = torch.Tensor([[1, 1], [2, 2]])
data['dev'].x = torch.Tensor([[1], [2]])
```

4. Теперь создаем соединения между нашими узлами. Связи имеют различный смысл, поэтому создадим три набора индексов ребер. Мы

объявляем каждый набор, используя триплет (тип узла-источника, тип ребра, тип узла-назначения), например data['user', 'follows', 'user'].edge_index:

```
# [2, num_edges_follows]
data['user', 'follows', 'user'].edge_index = torch.Tensor([[0, 1],
                                                           [1, 2]])
data['user', 'plays', 'game'].edge_index = torch.Tensor([[0, 1, 1, 2],
                                                         [0, 0, 1, 1]])
data['dev', 'develops', 'game'].edge_index = torch.Tensor([[0, 1],
                                                           [0, 1]])
```

5. У ребер тоже могут быть признаки. Например, ребра plays могут включать количество часов, которые пользователь провел, играя в соответствующую игру. В следующем примере предполагается, что пользователь 1 играл в игру 1 в течение 2 часов, пользователь 2 играл в игру 1 полчаса и в игру 2 в течение 10 часов, а пользователь 3 играл в игру 2 в течение 12 часов:

```
data['user', 'plays', 'game'].edge_attr = torch.Tensor(
    [[2], [0.5], [10], [12]]
)
```

6. Наконец, давайте выведем объект data, чтобы увидеть результат:

```
data
```

```
HeteroData(
  user={ x=[3, 4] },
  game={ x=[2, 2] },
  dev={ x=[2, 1] },
  (user, follows, user)={ edge_index=[2, 2] },
  (user, plays, game)={
    edge_index=[2, 4],
    edge_attr=[4, 1]
  },
  (dev, develops, game)={ edge_index=[2, 2] }
)
```

Как видно из этой реализации, разные типы узлов и ребер не могут использовать тензоры одного и того же размера. По сути, это невозможно, поскольку у разных типов узлов и ребер будут разные размерности. Возникает новая проблема – как мы можем агрегировать информацию из нескольких тензоров с помощью графовых нейронных сетей (GNN)?

До сих пор мы рассматривали только один тип узлов и один тип ребер. На практике матрица весов настраивается под конкретную размерность входных данных. Однако как мы можем реализовать GNN, когда у нас входные данные разных размерностей?

Преобразование гомогенных GNN в гетерогенные GNN

Чтобы лучше понять проблему, давайте рассмотрим реальный набор данных в качестве примера. В рамках проекта Библиография по компьютерным наукам DBLP сформирован набор данных [2-3], который содержит четыре типа узлов – статьи (14 328), термины (7723), авторы (4057) и конференции (20). Цель этого набора данных – правильно классифицировать авторов по четырем категориям: database (базы данных), data mining (добыча данных), artificial intelligence (искусственный интеллект) и information retrieval (информационный поиск). Признаки узлов авторов представляют собой «мешок слов» (`0` или `1`) из 334 ключевых слов, которые они могли использовать в своих публикациях. На следующем рисунке представлены отношения между разными типами узлов.

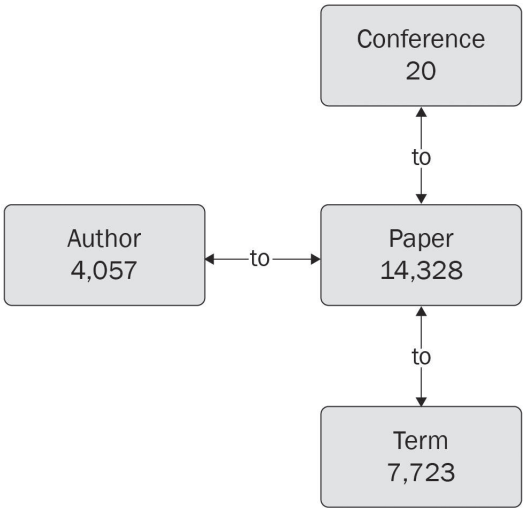


Рис. 12.3 ❖ Отношения между типами узлов в наборе данных DBLP

У этих типов узлов нет одинаковой размерности и семантических отношений. В гетерогенных графах отношения между узлами очень важны, поэтому нам важно рассматривать именно пары узлов. Например, вместо того чтобы просто подавать узлы-авторы в GNN-слой, нужно рассматривать пары типа (`автор, статья`). Это означает, что теперь нам нужен GNN-слой для каждого отношения, в данном случае «to»-отношения являются двунаправленными, поэтому мы получим шесть слоев.

У этих новых слоев будут независимые матрицы весов с правильным размером для каждого типа узлов. К сожалению, мы решили только половину проблемы. Действительно теперь у нас есть шесть разных слоев, у которых нет общей информации. Мы можем исправить это, введя **связи с пропусками** (skip-connections), **общие слои** (shared layers), **прыгающие знания** (jumping knowledge) и т. д. [4].

Прежде чем преобразовывать гомогенную модель в гетерогенную, давайте реализуем классическую графовую нейронную сеть с механизмом внимания (GAT) для набора данных DBLP. GAT не может учитывать различные отношения, поэтому мы должны передать ей уникальную матрицу смежности, связывающую авторов друг с другом. К счастью, теперь у нас есть метод, позволяющий легко сгенерировать такую матрицу смежности, – мы можем создать метапуть, например `автор-статья-автор`, который будет связывать авторов из одних и тех же статей.

Примечание

Кроме того, мы можем построить матрицу смежности с помощью случайных блужданий. Даже если граф гетерогенен, мы можем исследовать его и соединить узлы, которые часто появляются в одних и тех же последовательностях.

1. Мы импортируем необходимые библиотеки:

```
from torch import nn
import torch.nn.functional as F

import torch_geometric.transforms as T
from torch_geometric.datasets import DBLP
from torch_geometric.nn import GAT
```

2. Задаем метапуть, который будем использовать в дальнейшем, используя следующий синтаксис:

```
metapaths = [[('author', 'paper'), ('paper', 'author')]]
```

3. Мы используем функцию преобразования `AddMetaPaths` для автоматического вычисления нашего метапути. Затем задаем `drop_orig_edge_types=True`, чтобы удалить другие отношения из набора данных (GAT может рассматривать только одно отношение):

```
transform = T.AddMetaPaths(metapaths=metapaths,
                           drop_orig_edge_types=True)
```

4. Загружаем набор данных и печатаем информацию о нем. Обратите внимание на отношение (`author, metapath_0, author`), созданное с помощью нашей функции преобразования:

```
dataset = DBLP('.', transform=transform)
data = dataset[0]
```

```
print(data)

HeteroData(
  metapath_dict={ (author, metapath_0, author)=[2] },
  author={
    x=[4057, 334],
    y=[4057],
    train_mask=[4057],
    val_mask=[4057],
    test_mask=[4057]
  },
  paper={ x=[14328, 4231] },
  term={ x=[7723, 50] },
  conference={ num_nodes=20 },
  (author, metapath_0, author)={ edge_index=[2, 11113] }
)
```

5. Мы напрямую создаем модель GAT с одним слоем, задав `in_channels=-1` для ленивой инициализации (значение `in_channels` будет вычислено автоматически на основе входных данных) и `out_channels=4`, так как нам нужно классифицировать узлы-авторы на четыре категории:

```
set_seed()

model = GAT(in_channels=-1,
            hidden_channels=64,
            out_channels=4,
            num_layers=1)
```

6. Мы используем оптимизатор Adam и помещаем модель и данные на GPU, если это возможно:

```
optimizer = torch.optim.Adam(model.parameters(),
                             lr=0.001,
                             weight_decay=0.001)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
data, model = data.to(device), model.to(device)
```

7. Функция `test()` измеряет качество модели:

```
@torch.no_grad()
def test(mask):
    model.eval()
    pred = model(data.x_dict['author'],
                 data.edge_index_dict[
                     ('author', 'metapath_0', 'author')
                 ]).argmax(dim=-1)
    acc = (pred[mask] == data['author'].y[mask]).sum() / mask.sum()
    return float(acc)
```

8. Мы создаем классический цикл обучения, в котором осуществляется тщательный отбор признаков узлов (`author`) и индексов ребер (`author`, `metapath_0` и `author`):

```
for epoch in range(101):
    model.train()
    optimizer.zero_grad()
    out = model(data.x_dict['author'],
                data.edge_index_dict[
                    ('author', 'metapath_0', 'author')
                ])
    mask = data['author'].train_mask
    loss = F.cross_entropy(out[mask],
                           data['author'].y[mask])
    loss.backward()
    optimizer.step()

    if epoch % 20 == 0:
        train_acc = test(data['author'].train_mask)
        val_acc = test(data['author'].val_mask)
        print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: '
              f'{loss:.4f} | Правильность на обуч. выборке: '
              f'{train_acc*100:.2f}% \n| Правильность на валид. '
              f'выборке: {val_acc*100:.2f}%')
```

```
Epoch:   0 | Train Loss: 1.4351 | Train Acc: 25.25% | Val Acc: 22.00%
Epoch:  20 | Train Loss: 1.2815 | Train Acc: 46.50% | Val Acc: 37.50%
Epoch:  40 | Train Loss: 1.1641 | Train Acc: 63.75% | Val Acc: 53.25%
Epoch:  60 | Train Loss: 1.0628 | Train Acc: 76.50% | Val Acc: 63.25%
Epoch:  80 | Train Loss: 0.9771 | Train Acc: 81.00% | Val Acc: 66.25%
Epoch: 100 | Train Loss: 0.9040 | Train Acc: 83.50% | Val Acc: 67.75%
```

9. Мы вычисляем качество модели на тестовом наборе:

```
test_acc = test(data['author'].test_mask)
print(f'Качество на тестовом наборе: {test_acc*100:.2f}%')
```

```
Качество на тестовом наборе: 72.43%
```

Мы свели наш гетерогенный набор данных к гомогенному, используя метапуть, и применили традиционную модель GAT. Мы получили правильность 72.43 % на тестовом наборе, что дает хорошую отправную точку для сравнения с другими методами.

Теперь давайте создадим гетерогенную версию этой модели GAT. Следуя ранее описанному методу, нам необходимо шесть GAT-слоев вместо одного. Однако не нужно делать это вручную, так как PyTorch Geometric может сделать это автоматически с помощью функций `to_hetero()` или `to_hetero_bases()`. Функция `to_hetero()` принимает три важных параметра:

- `module`: гомогенная модель, которую мы хотим преобразовать;
- `metadata`: информация о гетерогенной природе графа, представленная в виде кортежа (`node_types`, `edge_types`);
- `aggr`: агрегатор для объединения эмбеддингов узлов, сгенерированных разными отношениями (например, `sum`, `max` или `mean`).

На рис. 12.4 показана наша гомогенная модель GAT (слева) и ее гетерогенный вариант (справа), полученный с помощью функции `to_hetero()`.

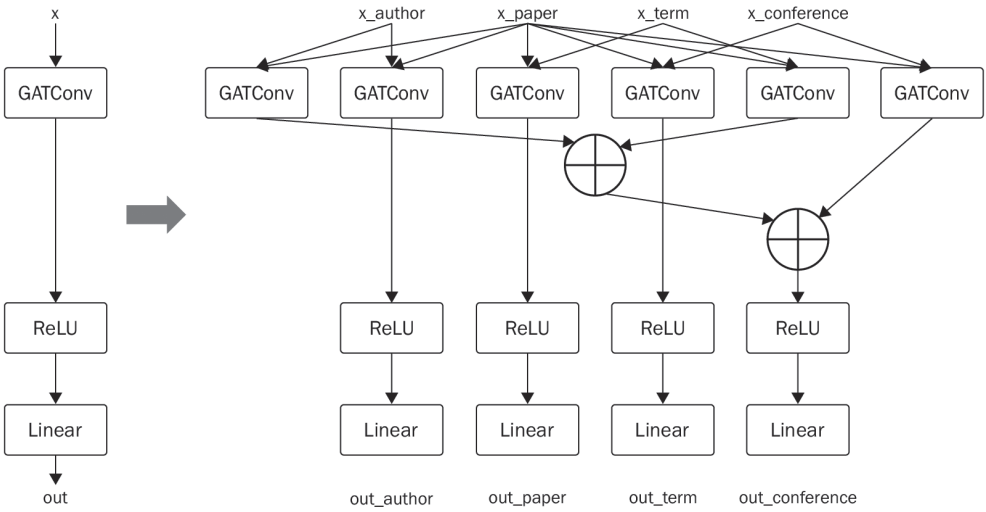


Рис. 12.4 ❖ Архитектура гомогенной модели GAT (слева) и гетерогенной модели GAT (справа) для набора DBLP

Как мы увидим далее, реализация гетерогенной модели довольна схожа с реализацией гомогенной модели.

1. Сначала мы импортируем GNN-слои из PyTorch Geometric:

```
from torch_geometric.nn import GATConv, Linear, to_hetero
```

2. Загружаем набор данных DBLP:

```
dataset = DBLP(root='.')
data = dataset[0]
```

3. Когда мы вывели информацию об этом наборе данных, вы могли заметить, что у узлов-конференций нет никаких признаков. Это проблема, поскольку наша архитектура предполагает, что у каждого типа узла есть свои собственные признаки. Мы можем решить эту проблему, генерируя нулевые значения в качестве признаков следующим образом:

```
data['conference'].x = torch.zeros(20, 1)
```

4. Мы создаем свой собственный класс GAT с использованием GAT-слоев и линейных слоев. Обратите внимание, что мы снова используем ленивую инициализацию с помощью кортежа (-1, -1):

```
class GAT(torch.nn.Module):
    def __init__(self, dim_h, dim_out):
        super().__init__()
        self.conv = GATConv((-1, -1), dim_h, add_self_loops=False)
        self.linear = nn.Linear(dim_h, dim_out)

    def forward(self, x, edge_index):
```

```
        h = self.conv(x, edge_index).relu()
        h = self.linear(h)
        return h
```

5. Мы создаем экземпляр класса `GAT` и преобразовываем традиционную модель в гетерогенную с помощью функции `to_hetero()`:

```
model = GAT(dim_h=64, dim_out=4)
model = to_hetero(model, data.metadata(), aggr='sum')
print(model)
```

```
GraphModule(
  (conv): ModuleDict(
    (author__to__paper): GATConv((-1, -1), 64, heads=1)
    (paper__to__author): GATConv((-1, -1), 64, heads=1)
    (paper__to__term): GATConv((-1, -1), 64, heads=1)
    (paper__to__conference): GATConv((-1, -1), 64, heads=1)
    (term__to__paper): GATConv((-1, -1), 64, heads=1)
    (conference__to__paper): GATConv((-1, -1), 64, heads=1)
  )
  (linear): ModuleDict(
    (author): Linear(in_features=64, out_features=4, bias=True)
    (paper): Linear(in_features=64, out_features=4, bias=True)
    (term): Linear(in_features=64, out_features=4, bias=True)
    (conference): Linear(in_features=64, out_features=4, bias=True)
  )
)
```

6. Используем оптимизатор Adam и помещаем модель и данные на GPU, если это возможно:

```
set_seed()

optimizer = torch.optim.Adam(model.parameters(),
                             lr=0.001,
                             weight_decay=0.001)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
data, model = data.to(device), model.to(device)
```

7. Процесс тестирования очень похож. В этот раз нам не нужно указывать какое-либо отношение, поскольку модель учитывает все отношения:

```
@torch.no_grad()
def test(mask):
    model.eval()
    pred = model(data.x_dict,
                 data.edge_index_dict)['author'].argmax(dim=-1)
    acc = (pred[mask] == data['author'].y[mask]).sum() / mask.sum()
    return float(acc)
```

8. Теперь создаем цикл обучения:

```
for epoch in range(101):
    model.train()
    optimizer.zero_grad()
    out = model(data.x_dict, data.edge_index_dict)['author']
    mask = data['author'].train_mask
    loss = F.cross_entropy(out[mask], data['author'].y[mask])
    loss.backward()
    optimizer.step()

    if epoch % 20 == 0:
        train_acc = test(data['author'].train_mask)
        val_acc = test(data['author'].val_mask)
        print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: '
              f'{loss:.4f} | Правильность на обуч. выборке: '
              f'{train_acc*100:.2f}% \n| Правильность на валид. '
              f'выборке: {val_acc*100:.2f}%')
```

```
Эпоха   0:
| Функция потерь на обуч. выборке: 1.3974 | Правильность на обуч. выборке: 20.75%
| Правильность на валид. выборке: 23.00%
Эпоха  20:
| Функция потерь на обуч. выборке: 1.2047 | Правильность на обуч. выборке: 95.25%
| Правильность на валид. выборке: 68.00%
Эпоха  40:
| Функция потерь на обуч. выборке: 0.8654 | Правильность на обуч. выборке: 96.75%
| Правильность на валид. выборке: 67.50%
Эпоха  60:
| Функция потерь на обуч. выборке: 0.5061 | Правильность на обуч. выборке: 98.75%
| Правильность на валид. выборке: 73.50%
Эпоха  80:
| Функция потерь на обуч. выборке: 0.2580 | Правильность на обуч. выборке: 99.50%
| Правильность на валид. выборке: 73.50%
Эпоха 100:
| Функция потерь на обуч. выборке: 0.1384 | Правильность на обуч. выборке: 100.00%
| Правильность на валид. выборке: 74.00%
```

9. Оцениваем качество на тестовом наборе:

```
test_acc = test(data['author'].test_mask)
print(f'Качество на тестовом наборе: {test_acc*100:.2f}%')
```

```
Качество на тестовом наборе: 78.63%
```

Гетерогенная модель GAT демонстрирует правильность 78.42 % на тестовом наборе. Это хорошее улучшение (+6.2 %) по сравнению с гомогенной версией, но можно ли добиться еще лучших результатов? В следующем разделе мы исследуем архитектуру, специально разработанную для обработки гетерогенных сетей.

Реализация иерархической нейронной сети с самовниманием

В данном разделе мы реализуем модель GNN, предназначенную для работы с гетерогенными графами, – **иерархическую нейронную сеть с самовниманием** (hierarchical self-attention network – HAN). Эта архитектура была представлена Лю Цзинь совместно с коллегами в 2021 году [5]. HAN использует самовнимание на двух разных уровнях:

- **внимание на уровне узлов** (node-level attention) для понимания важности соседних узлов в заданном метапути (типа гомогенной GAT);
- **внимание на семантическом уровне** (semantic-level attention) для изучения важности каждого метапути. Это главная отличительная особенность HAN, которая позволяет автоматически выбирать лучшие метапути для задачи; например, в некоторых задачах типа прогнозирования количества игроков метапуть `игра-пользователь-игра` будет, возможно, более релевантным, чем `игра-разработчик-игра`.

В следующем разделе мы подробно рассмотрим три основных компонента – внимание на уровне узлов, внимание на семантическом уровне и модуль прогнозов. Эта архитектура проиллюстрирована на рис. 12.5.

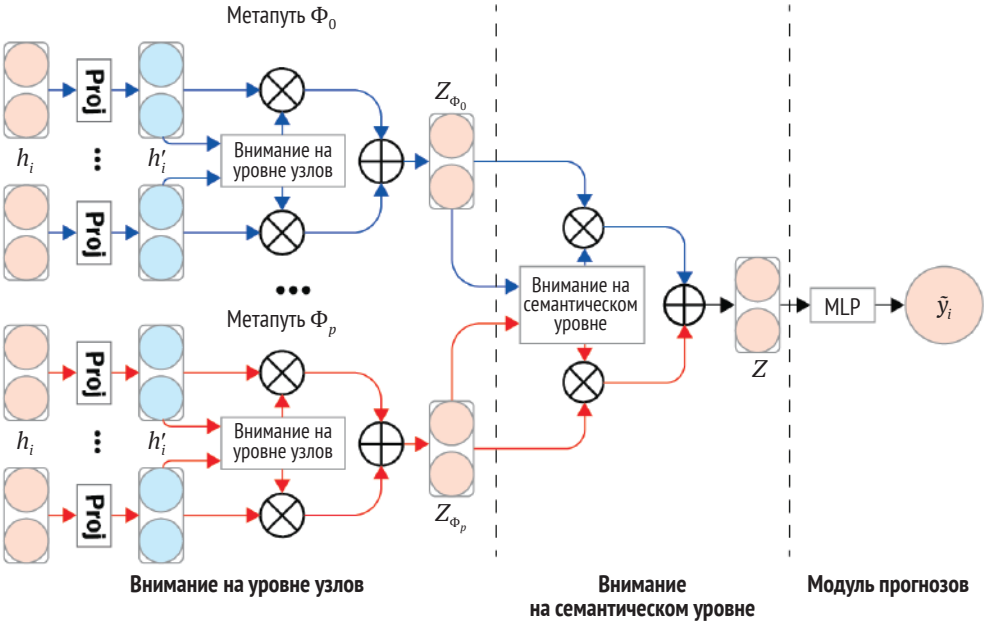


Рис. 12.5 ❖ Архитектура HAN с тремя основными модулями

Как и в GAT, первый этап включает в себя проецирование узлов в объединенное пространство признаков для каждого метапути. Затем мы вычисляем вес пары узлов (конкатенации двух спроецированных узлов) в том же метапути с помощью второй матрицы весов. К этому результату применяется нелинейная функция, после чего происходит нормализация с использованием функции softmax. Нормализованная оценка внимания (важность) узла *j* для узла *i* вычисляется следующим образом:

$$\alpha_{ij}^{\Phi} = \frac{\exp(\sigma(\boldsymbol{a}_{\Phi}^{T}[\boldsymbol{W}_{\Phi}\mathbf{h}_i \| \boldsymbol{W}_{\Phi}\mathbf{h}_j]))}{\sum_{k\in\mathcal{N}_i^{\Phi}} \exp(\sigma(\boldsymbol{a}_{\Phi}^{T}[\boldsymbol{W}_{\Phi}\mathbf{h}_i \| \boldsymbol{W}_{\Phi}\mathbf{h}_k]))}.$$

Здесь h_i обозначает признаки узла *i*, W_{Φ} – это общая матрица весов для метапути Ф, α_{Φ} – матрица весов внимания для метапути Ф, σ – это нелинейная функция активации (например, LeakyReLU), а $\mathcal{N}_i^{\Phi}$ – это множество соседей узла *i* (включая сам узел) в метапути Ф.

Кроме того, применяется многоголовое внимание для получения итогового эмбеддинга:

$$z_i = \Big\|_{k=1}^{K} \sigma\Big(\sum_{j\in\mathcal{N}_i} \alpha_{i,j}^{\Phi} \cdot W_{\Phi} h_j\Big).$$

С помощью внимания на семантическом уровне мы повторяем аналогичную операцию с оценками внимания для каждого метапути (обозначаются $\beta_{\Phi_1}, \beta_{\Phi_2}, \ldots, \beta_{\Phi_p}$). Каждый эмбеддинг узла в данном метапути (обозначается как Z_{Φ_p}) подается в многослойный перцептрон (MLP), который применяет нелинейное преобразование. Мы сравниваем этот результат с новым вектором внимания *q*, выступающим в качестве меры сходства. Мы усредняем этот результат для вычисления важности данного метапути:

$$w_{\Phi_p} = \frac{1}{|V|}\sum_{i\in V} q^{T} \cdot \tanh(W \cdot z_i^{\Phi_p} + b).$$

Здесь *W* (матрица весов MLP), *b* (смещение MLP) и *q* (вектор внимания на семантическом уровне) являются общими для всех метапутей.

Этот результат необходимо нормализовать для сравнения различных оценок внимания на семантическом уровне. Мы используем функцию softmax для получения наших итоговых весов:

$$\beta_{\Phi_p} = \frac{\exp(w_{\Phi_p})}{\sum_{k=1}^{P} \exp(w_{\Phi_k})}.$$

Итоговый эмбеддинг *Z*, объединяющий в себе внимание на уровне узлов и внимание на семантическом уровне, мы получаем следующим образом:

$$Z = \sum_{p=1}^{P} \beta_{\Phi_p} \cdot Z_{\Phi_p}.$$

Затем применяется итоговый слой типа MLP для тонкой настройки модели под конкретную задачу, например под классификацию узлов или прогнозирование связей.

Давайте реализуем эту архитектуру в PyTorch Geometric для набора данных DBLP.

1. Сначала импортируем HAN-слой:

```
set_seed()

from torch_geometric.nn import HANConv
```

2. Мы загружаем набор данных `DBLP` и задаем дамми-переменные для узлов-конференций:

```
dataset = DBLP(root='.')
data = dataset[0]

data['conference'].x = torch.zeros(20, 1)
```

3. Создаем класс `HAN` с двумя слоями – слоем сверки HAN (`HANConv`) и линейным слоем (`nn.Linear`) для итоговой классификации:

```
class HAN(nn.Module):
    def __init__(self, dim_in, dim_out, dim_h=128, heads=8):
        super().__init__()
        self.han = HANConv(dim_in, dim_h, heads=heads,
                           dropout=0.6, metadata=data.metadata())
        self.linear = nn.Linear(dim_h, dim_out)
```

4. В функции `forward()` необходимо указать, что нас интересуют авторы:

```
    def forward(self, x_dict, edge_index_dict):
        out = self.han(x_dict, edge_index_dict)
        out = self.linear(out['author'])
        return out
```

5. Мы создаем экземпляр класса `HAN` с ленивой инициализацией (`dim_in=-1`), поэтому PyTorch Geometric автоматически вычисляет размер входных данных для каждого типа узла:

```
model = HAN(dim_in=-1, dim_out=4)
```

6. Используем оптимизатор Adam и помещаем модель и данные на GPU, если это возможно:

```
optimizer = torch.optim.Adam(model.parameters(),
                             lr=0.001,
                             weight_decay=0.001)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
data, model = data.to(device), model.to(device)
```

7. Функция `test()` вычисляет правильность для нашей задачи классификации:

```
@torch.no_grad()
def test(mask):
    model.eval()
    pred = model(data.x_dict, data.edge_index_dict).argmax(dim=-1)
    acc = (pred[mask] == data['author'].y[mask]).sum() / mask.sum()
    return float(acc)
```

8. Мы обучаем модель, задав 100 эпох. Единственное отличие от цикла обучения для гомогенной графовой нейронной сети – нужно указать, что нас интересует тип узла «автор»:

```
for epoch in range(101):
    model.train()
    optimizer.zero_grad()
    out = model(data.x_dict, data.edge_index_dict)
    mask = data['author'].train_mask
    loss = F.cross_entropy(out[mask], data['author'].y[mask])
    loss.backward()
    optimizer.step()

    if epoch % 20 == 0:
        train_acc = test(data['author'].train_mask)
        val_acc = test(data['author'].val_mask)
        print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. выборке: '
              f'{loss:.4f} | Правильность на обуч. выборке: '
              f'{train_acc*100:.2f}% \n| Правильность на валид. '
              f'выборке: {val_acc*100:.2f}%')
```

9. Оцениваем качество модели на тестовом наборе:

```
test_acc = test(data['author'].test_mask)
print(f'Качество на тестовом наборе: {test_acc*100:.2f}%')
```

```
Качество на тестовом наборе: 81.58%
```

HAN демонстрирует правильность 81.58 % на тестовом наборе, что выше значения правильности при использовании гетерогенной модели GAT (78.63 %) и классической модели GAT (72.43 %). Это показывает важность построения качественных представлений, объединяющих различные типы узлов и отношений. Методы гетерогенных графов сильно зависят от области применения, но стоит попробовать разные варианты, особенно когда отношения в сети имеют содержательный смысл.

Выводы

В этой главе мы представили **нейронную сеть передачи сообщений** (Message Passing Neural Network – MPNN, или MP-GNN), позволяющую обобщить GNN-слои с помощью трех операций – message, aggregate и update. Затем расширили ее применение на гетерогенные сети, состоящие из различных

типов узлов и ребер. Такой вид графа позволяет нам представлять различные отношения между сущностями, что более информативно, чем использование одного типа связи.

Кроме того, мы посмотрели, как преобразовать гомогенные GNN в гетерогенные с помощью PyTorch Geometric. Мы описали различные слои нашей гетерогенной модели GAT, которые принимают пары узлов в качестве входных данных для моделирования их отношений. Наконец, мы реализовали гетерогенную архитектуру с HAN и сравнили результаты трех методов на наборе данных DBLP. Это подтвердило важность использования гетерогенной информации, представленной в этом типе сети.

В главе 13 «Темпоральные графовые нейронные сети» мы выясним, как учитывать время в графовых нейронных сетях (GNN). Темпоральные графы откроют для вас различные примеры применения типа прогнозирование дорожного трафика. Кроме того, будет представлено расширение библиотеки PyG под названием PyTorch Geometric Temporal, которое поможет нам реализовать новые модели, специально разработанные для работы с временем.

Дополнительное чтение

[1] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. *Neural Message Passing for Quantum Chemistry*. arXiv, 2017. DOI: 10.48550/ARXIV.1704.01212. Доступ по ссылке https://arxiv.org/abs/1704.01212.

[2] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. *ArnetMiner: Extraction and Mining of Academic Social Networks*. In Proceedings of the Fourteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'2008). pp.990–998. Доступ по ссылке https://dl.acm.org/doi/abs/10.1145/1401890.1402008.

[3] X. Fu, J. Zhang, Z. Meng, and I. King. *MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding*. Apr. 2020. DOI: 10.1145/3366423.3380297. Доступ по ссылке https://arxiv.org/abs/2002.01680.

[4] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. *Modeling Relational Data with Graph Convolutional Networks*. arXiv, 2017. DOI: 10.48550/ARXIV.1703.06103. Доступ по ссылке https://arxiv.org/abs/1703.06103.

[5] J. Liu, Y. Wang, S. Xiang, and C. Pan. *HAN: An Efficient Hierarchical Self-Attention Network for Skeleton-Based Gesture Recognition*. arXiv, 2021. DOI: 10.48550/ARXIV.2106.13391. Доступ по ссылке https://arxiv.org/abs/2106.13391.

Глава 13

Темпоральные графовые нейронные сети

В предыдущих главах мы рассматривали графы, в которых ребра и признаки не меняются. Однако в реальном мире существует множество областей, где это не так. Например, в социальных сетях люди подписываются и отписываются от других пользователей, посты становятся вирусными, а профили меняются со временем. Эту динамичность нельзя представить с помощью ранее описанных архитектур графовых нейронных сетей (GNN). Вместо этого мы должны внедрить новое, темпоральное (или временно́е) измерение, чтобы превратить статические графы в динамические. Эти динамические сети будут использоваться в качестве входных данных для нового семейства GNN: **темпоральных графовых нейронных сетей** (Temporal Graph Neural Networks – T-GNN), также называемых **пространственно-темпоральными графовыми нейронными сетями** (Spatio-Temporal Graph Neural Networks).

В этой главе мы рассмотрим два вида **динамических графов** (dynamic graphs), содержащих информацию о пространстве и времени. Мы перечислим различные области применения и сосредоточимся на прогнозировании временных рядов, где в основном применяются темпоральные GNN. Второй раздел посвящен задаче, которую мы рассматривали ранее, – прогнозированию веб-трафика. На этот раз мы будем использовать темпоральную информацию для улучшения результатов и получения надежных прогнозов. Наконец, мы опишем еще одну архитектуру темпоральных GNN, разработанную для динамических графов, и применим ее для прогнозирования количества случаев COVID-19 в различных регионах Англии.

К концу этой главы вы узнаете разницу между двумя основными типами динамических графов. Это знание особенно пригодится, когда нужно будет выбрать соответствующий тип графа для моделирования ваших данных. Кроме того, вы узнаете о проектировании и архитектуре двух темпоральных

GNN, а также о том, как их реализовать с помощью PyTorch Geometric Temporal. Этот этап позволит вам решать задачи с использованием темпоральной информации.

В этой главе мы рассмотрим следующие основные темы:

- «Знакомство с динамическими графами»,
- «Прогнозирование веб-трафика»,
- «Прогнозирование случаев COVID-19».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter13.

Знакомство с динамическими графами

Динамические графы и темпоральные графовые нейронные сети открывают широкий спектр новых задач, таких как прогнозирование транспортного трафика и веб-трафика, классификация движений, эпидемиологическое прогнозирование, прогнозирование связей, прогнозирование энергопотребления и т. д. Для прогнозирования временных рядов особенно популярен именно этот тип графов, так как мы можем использовать исторические данные для прогнозирования будущего поведения системы.

В этой главе мы сосредоточимся на графах с темпоральной компонентой. Их можно разбить на две категории:

- **статические графы с темпоральными сигналами** (static graphs with temporal signals): соответствующий граф не изменяется, но характеристики и метки меняются со временем;
- **динамические графы с темпоральными сигналами** (dynamic graphs with temporal signals): топология графа (наличие узлов и ребер), характеристики и метки изменяются со временем.

В первом случае топология графа *статична*. Например, она может представлять собой сеть городов в стране для прогнозирования трафика: характеристики меняются со временем, но связи остаются прежними.

Во втором варианте узлы и/или связи *динамичны*. Это полезно для представления социальной сети, где связи между пользователями могут появляться или исчезать со временем. Этот вариант более распространен, но и более сложен для изучения и реализации.

В следующих разделах мы рассмотрим, как обрабатывать эти два типа графов с темпоральными сигналами с использованием PyTorch Geometric Temporal.

Прогнозирование веб-трафика

В этом разделе мы будем предсказывать веб-трафик статей «Википедии» (как пример статического графа с темпоральным сигналом), используя темпоральную GNN. Эта задача регрессии уже рассматривалась в главе 6 «Знакомство с графовыми сверточными нейронными сетями». Однако в том варианте задачи мы выполняли прогнозирование трафика с использованием статического набора данных без темпорального сигнала: наша модель не имела информации о наблюдениях в прошлые моменты времени. И это было проблемой, потому что модель не могла определить, увеличивается или уменьшается трафик. Теперь мы можем улучшить эту модель, включив информацию о прошлом.

Сначала познакомимся с двумя вариантами архитектуры темпоральной GNN, а затем реализуем эту архитектуру, используя PyTorch Geometric Temporal.

Знакомство с EvolveGCN

Для выполнения этой задачи воспользуемся архитектурой EvolveGCN, которую в 2019 году представил Альдо Парейя совместно с коллегами [1]. Она предлагает естественное сочетание графовых нейронных сетей (GNN) и рекуррентных нейронных сетей (RNN). Предыдущие подходы, такие как графовые сверточные рекуррентные сети, применяли RNN вместе с операторами графовых сверток для вычисления эмбеддингов узлов. В отличие от них EvolveGCN применяет RNN к самим параметрам GCN. Как следует из названия, GCN меняются (эволюционируют) со временем, чтобы создавать релевантные темпоральные эмбеддинги узлов. Ниже на рис. 13.1 дано общее представление этого процесса.

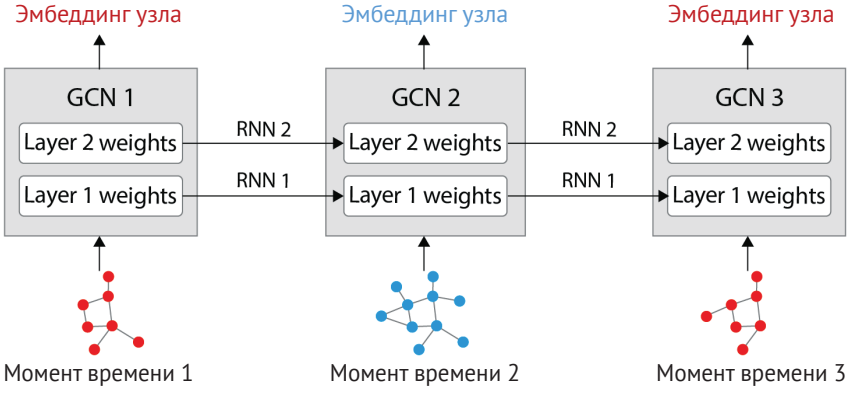


Рис. 13.1 ❖ Архитектура EvolveGCN, генерирующая эмбеддинги узлов для статического или динамического графа с темпоральным сигналом

Эта архитектура имеет два варианта:

- **EvolveGCN-H**, где рекуррентная нейронная сеть учитывает как предыдущие параметры GCN, так и текущие эмбеддинги узлов;
- **EvolveGCN-O**, где рекуррентная нейронная сеть учитывает только предыдущие параметры GCN.

EvolveGCN-H обычно использует **управляемые реккурентные блоки** (Gated Recurrent Units – GRU) вместо обычной RNN. GRU представляет собой упрощенную версию **нейронной сети с долговременной краткосрочной памятью** (Long Short-Term Memory – LSTM), которая достигает сопоставимого качества с меньшим количеством параметров. Она состоит из вентилей сброса, вентиля обновления и ячейки состояния. В этой архитектуре GRU обновляет матрицу весов GCN для слоя l в момент времени t следующим образом:

$$W_t^{(l)} = \text{GRU}(H_t^{(l)}, W_{t-1}^{(l)}),$$

где $H_t^{(l)}$ обозначает эмбеддинги узлов, созданные для слоя l и момента времени t, а $W_{t-1}^{(l)}$ – это матрица весов для слоя l с предыдущего временного шага.

Эта результирующая матрица весов GCN затем используется для вычисления эмбеддингов узлов следующего слоя:

$$\begin{aligned} H_t^{(l+1)} &= \text{GCN}(A_t, H_t^{(l)}, W_t^{(t)}) \\ &= \tilde{D}^{-\frac{1}{2}} \tilde{A}^T \tilde{D}^{-\frac{1}{2}} H_t^{(l)} W_t^{(t)^T}. \end{aligned}$$

Здесь $\tilde{A}$ – это матрица смежности, включая самопетли, а $\tilde{D}$ – это матрица степеней с самопетлями. Эти шаги кратко представлены на рис. 13.2.

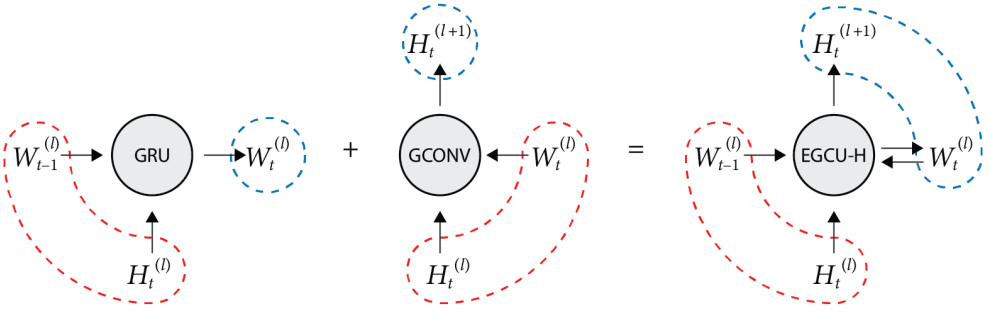


Рис. 13.2 ❖ Архитектура EvolveGCN-H с GRU и GNN

EvolveGCN-H можно реализовать с помощью управляемых реккурентных блоков (GRU), которые получают два расширения:

- входные и скрытые состояния представлены матрицами вместо векторов с целью корректного хранения матриц весов GCN;

- количество столбцов входных данных должно совпадать с количеством столбцов скрытого состояния.

Эти расширения не требуются для варианта EvolveGCN-O. Действительно EvolveGCN-O основана на сети LSTM для моделирования отношений «вход-выход». Нам не нужно передавать скрытое состояние в LSTM, поскольку она уже включает ячейку, которая помнит предыдущие значения. Этот механизм упрощает шаг обновления и может быть записан следующим образом:

$$W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)}).$$

Результирующая матрица весов GCN используется таким же образом для вычисления эмбеддингов узлов следующего слоя:

$$\begin{aligned} H_t^{(l+1)} &= \text{GCN}(A_t, H_t^{(l)}, W_t^{(t)}) \\ &= \tilde{D}^{-\frac{1}{2}} \tilde{A}^T \tilde{D}^{-\frac{1}{2}} H_t^{(l)} W_t^{(t)^T}. \end{aligned}$$

Эта реализация проще, поскольку темпоральное измерение полностью зависит от обычной сети LSTM. На рис. 13.3 показано, как EvolveGCN-O обновляет матрицу весов $W_{t-1}^{(l)}$ и вычисляет эмбеддинги узлов $H_t^{(l+1)}$.

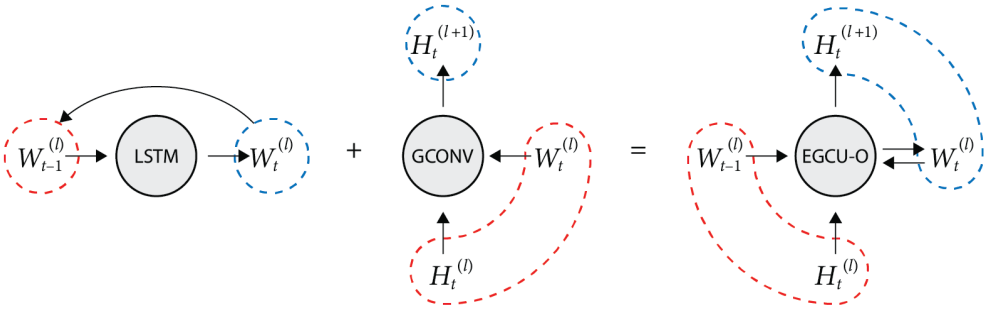


Рис. 13.3 ❖ Архитектура EvolveGCN-O с LSTM и GCN

Итак, какую же версию следует использовать? Как это часто бывает в машинном обучении, лучшее решение зависит от данных:

- EvolveGCN-H работает лучше, когда важны характеристики узлов, потому что здесь рекуррентная нейронная сеть явно включает в себя эмбеддинги узлов;
- EvolveGCN-O работает лучше, когда графовая структура играет важную роль, так как этот вариант больше фокусируется на топологических изменениях.

Следует отметить, что эти замечания в первую очередь носят теоретический характер, поэтому полезно протестировать оба варианта в ваших задачах. Это мы как раз и собираемся сейчас сделать, реализовав данные модели для прогнозирования веб-трафика.

Реализация EvolveGCN

В этом разделе мы хотим спрогнозировать веб-трафик для статического графа с временным сигналом. Набор данных WikiMaths состоит из 1068 статей, представленных в виде узлов. Признаки узлов соответствуют ежедневному количеству посещений (восемь признаков по умолчанию). Ребра взвешены, и веса представляют собой количество ссылок со страницы-источника на страницу-назначение. Мы хотим спрогнозировать ежедневное количество посещений пользователями данных страниц «Википедии» с 16 марта 2019 года по 15 марта 2021 года, что дает нам 731 контрольную точку (или 731 снапшот). Каждый снапшот – это граф, описывающий состояние системы в определенный момент времени. На рис. 13.4 приведена визуализация WikiMaths, созданная с помощью Gephi, в ней размер и цвет узлов пропорциональны их числу связей.

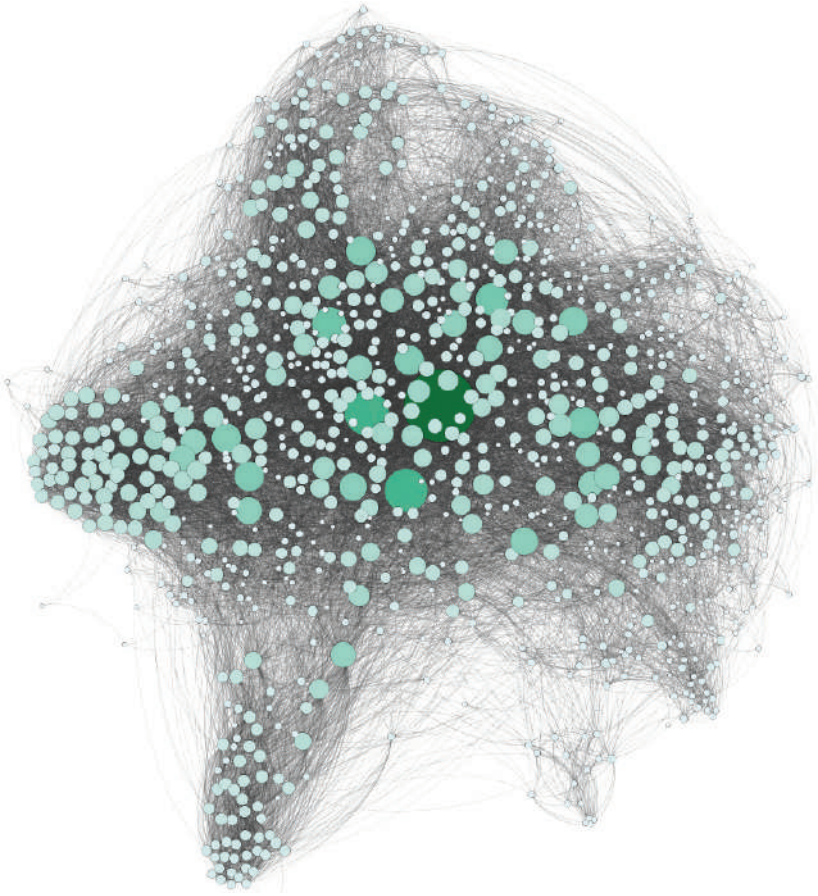


Рис. 13.4 ❖ Набор данных WikiMaths, представленный в виде невзвешенного графа (t=0)

PyTorch Geometric не поддерживает статические или динамические графы с темпоральным сигналом из-под коробки. К счастью, существует расширение под названием PyTorch Geometric Temporal [2], которое решает эту проблему, и в нем даже реализованы различные темпоральные GNN-слои. Во время разработки PyTorch Geometric Temporal также был обнародован набор данных WikiMaths. В этой главе мы будем использовать эту библиотеку для упрощения программного кода и разбора задач.

1. Нам нужно установить эту библиотеку:

```
!pip install torch-geometric-temporal==0.54.0
```

2. Мы импортируем класс `WikiMathDatasetLoader`, с помощью которого загрузим набор данных WikiMaths, импортируем функцию `temporal_signal_split()`, которая разобьет набор на обучающую и тестовую выборки с учетом временной структуры, а также импортируем класс `EvolveGCNH`:

```
from torch_geometric_temporal.signal import temporal_signal_split
from torch_geometric_temporal.dataset import WikiMathsDatasetLoader
from torch_geometric_temporal.nn.recurrent import EvolveGCNH
```

3. Загружаем набор данных WikiMaths, который является объектом `StaticGraphTemporalSignal`. В этом объекте `dataset[0]` описывает граф (также называемый в данном контексте снапшотом) в момент времени $t = 0$, а `dataset[500]` – граф в момент времени $t = 500$. Кроме того, мы разбиваем данные на обучающую и тестовую выборки с учетом временной структуры. Обучающий набор состоит из снапшотов, относящихся к более ранним временным периодам, в то время как тестовый набор содержит снапшоты более поздних периодов:

```
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm

dataset = WikiMathsDatasetLoader().get_dataset()
train_dataset, test_dataset = temporal_signal_split(
    dataset, train_ratio=0.5
)
```

4. Граф является статическим, поэтому количество узлов и ребер не изменяются. Тем не менее значения, содержащиеся в этих тензорах, различны. Сложно визуализировать значения каждого из 1068 узлов. Для более полного понимания этого набора данных мы можем вычислить средние и стандартные отклонения значений для каждого снапшота. Кроме того, полезно использовать скользящее среднее для сглаживания краткосрочных колебаний.

```
mean_cases = [snapshot.y.mean().item() for snapshot in dataset]
std_cases = [snapshot.y.std().item() for snapshot in dataset]
df = pd.DataFrame(mean_cases, columns=['mean'])
```

```
df['std'] = pd.DataFrame(std_cases, columns=['std'])
df['rolling'] = df['mean'].rolling(7).mean()
```

5. Визуализируем временной ряд:

```
plt.figure(figsize=(10,5))
plt.plot(df['mean'], 'k-', label='Среднее')
plt.plot(df['rolling'], 'g-', label='Скользящее среднее')
plt.grid(linestyle=':')
plt.fill_between(df.index,
                 df['mean'] - df['std'],
                 df['mean'] + df['std'],
                 color='r',
                 alpha=0.1)

plt.axvline(x=360, color='b', linestyle='--')

plt.text(360, 1.5, 'Разбиение\nобучение/тест',
         rotation=-90, color='b')

plt.xlabel('Время (дни)')
plt.ylabel('Нормализованное количество посещений')
plt.legend(loc='upper right');
```

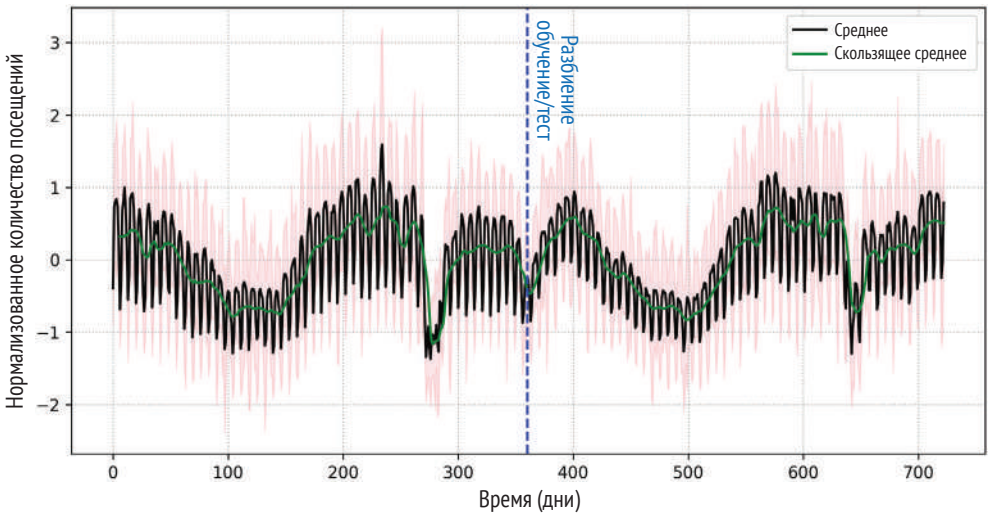


Рис. 13.5 ❖ Среднее нормализованное количество посещений (по данным набора WikiMaths) со скользящим средним

Наши данные содержат периодические паттерны, которые, возможно, можно исследовать с помощью темпоральной графовой нейронной сети (GNN). Теперь мы можем реализовать ее и посмотреть, как она работает.

6. Темпоральная графовая нейронная сеть (GNN) принимает в качестве входных параметров два параметра: количество узлов (`node_count`)

и размерность входных данных (dim_in). GNN состоит всего из двух слоев: слоя EvolveGCN-H и линейного слоя, который выводит предсказанное значение для каждого узла:

```
class TemporalGNN(torch.nn.Module):
    def __init__(self, node_count, dim_in):
        super().__init__()
        self.recurrent = EvolveGCNH(node_count, dim_in)
        self.linear = torch.nn.Linear(dim_in, 1)
```

7. Функция forward() применяет оба слоя к входящим данным, в качестве функции активации используем ReLU:

```
    def forward(self, x, edge_index, edge_weight):
        h = self.recurrent(x, edge_index, edge_weight).relu()
        h = self.linear(h)
        return h
```

8. Мы создаем экземпляр класса TemporalGNN и передаем ему количество узлов и размерность входных данных (размерность набора данных WikiMaths).

```
model = TemporalGNN(dataset[0].x.shape[0], dataset[0].x.shape[1])
model
```

```
TemporalGNN(
  (recurrent): EvolveGCNH(
    (pooling_layer): TopKPooling(8, ratio=0.00749063670411985, multiplier=1.0)
    (recurrent_layer): GRU(8, 8)
    (conv_layer): GCNConv_Fixed_W(8, 8)
  )
  (linear): Linear(in_features=8, out_features=1, bias=True)
)
```

Мы видим три слоя: TopKPooling, который обобщает входную матрицу в восемь столбцов, GRU, который обновляет весовую матрицу GCN, и GCNConv, который создает новое представление узла. Наконец, линейный слой выводит предсказанное значение для каждого узла в графе.

Мы будем обучать модель, используя оптимизатор Adam:

```
set_seed()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
model.train()
```

9. Создаем цикл обучения, который обучает модель на каждом снапшоте из обучающего набора. В нем же реализуем обратное распространение ошибки для каждого снапшота:

```
# Обучение
for epoch in tqdm(range(50)):
```

```
    for i, snapshot in enumerate(train_dataset):
        y_pred = model(snapshot.x,
                       snapshot.edge_index,
                       snapshot.edge_attr)
        loss = torch.mean((y_pred-snapshot.y)**2)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

10. Затем оцениваем качество модели на тестовой выборке. MSE усредняется по всей тестовой выборке для получения итоговой оценки:

```
# Оценка
model.eval()
loss = 0
for i, snapshot in enumerate(test_dataset):
    y_pred = model(snapshot.x,
                   snapshot.edge_index,
                   snapshot.edge_attr)
    mse = torch.mean((y_pred-snapshot.y)**2)
    loss += mse
loss = loss / (i+1)
print(f'MSE = {loss.item():.4f}')
```

```
MSE = 0.7676
```

11. В итоге получаем MSE 0.7676. Затем визуализируем средние значения, предсказанные нашей моделью для упрощения интерпретации. Процесс прост: нам нужно усреднить прогнозы, сохранить в списке и добавить на предыдущий график:

```
y_preds = [
    model(snapshot.x, snapshot.edge_index, snapshot.edge_attr)
    .squeeze()
    .detach()
    .numpy()
    .mean() for snapshot in test_dataset
]
```

```
plt.figure(figsize=(10,5), dpi=300)
plt.plot(df['mean'], 'k-', label='Среднее')
plt.plot(df['rolling'], 'g-', label='Скользящее среднее')
plt.plot(range(360,722), y_preds, 'r-', label='Прогноз')
plt.grid(linestyle=':')
plt.fill_between(df.index,
                 df['mean'] - df['std'],
                 df['mean'] + df['std'],
                 color='r',
                 alpha=0.1)

plt.axvline(x=360, color='b', linestyle='--')
```

```
plt.text(360, 1.5, 'Разбиение\nобучение/тест',
         rotation=-90, color='b')

plt.xlabel('Время (дни)')
plt.ylabel('Нормализованное количество посещений')
plt.legend(loc='upper right');
```

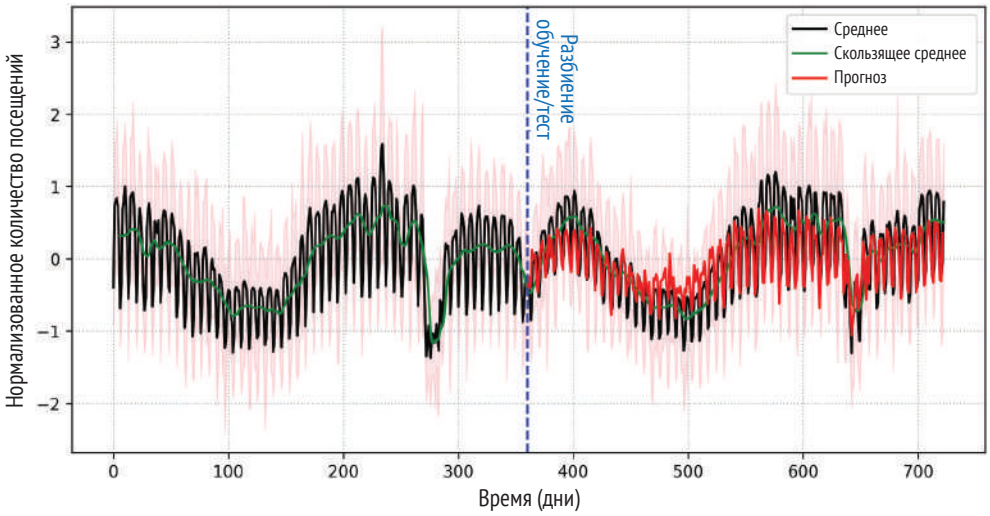


Рис. 13.6 ❖ Спрогнозированное среднее нормализованное количество посещений

Мы видим, что предсказанные значения следуют общему тренду в данных. Это отличный результат, учитывая ограниченный размер набора данных.

12. Наконец, давайте создадим диаграмму рассеяния, чтобы показать разницу между спрогнозированными и фактическими значениями для одного снапшота:

```
import seaborn as sns

y_pred = (
    model(test_dataset[0].x,
          test_dataset[0].edge_index,
          test_dataset[0].edge_attr)
    .detach()
    .squeeze()
    .numpy()
)

plt.figure(figsize=(10,5), dpi=300)
sns.regplot(x=test_dataset[0].y.numpy(), y=y_pred);
```

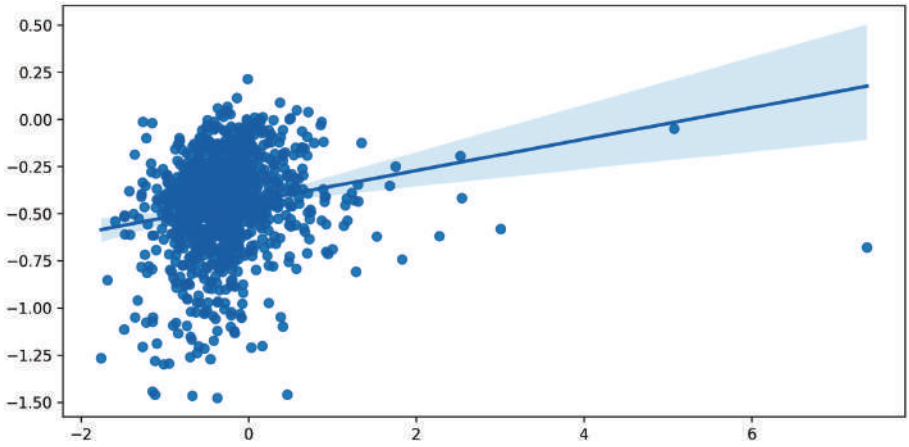


Рис. 13.7 ❖ Спрогнозированные и фактические значения для набора данных WikiMaths

Мы наблюдаем умеренную положительную корреляцию между спрогнозированными и фактическими значениями. Наша модель не является выдающейся с точки зрения правильности, но предыдущий график показал, что она хорошо понимает периодический характер данных.

Реализация варианта EvolveGCN-O во многом является аналогичной. Слой `EvolveGCNH` из PyTorch Geometric Temporal мы заменяем на `EvolveGCNO`. Этот слой не требует указания количества узлов, поэтому мы передаем только размерность входящих данных. Реализация выглядит следующим образом:

```
from torch_geometric_temporal.nn.recurrent import EvolveGCNO

class TemporalGNN(torch.nn.Module):
    def __init__(self, dim_in):
        super().__init__()
        self.recurrent = EvolveGCNO(dim_in, 1)
        self.linear = torch.nn.Linear(dim_in, 1)

    def forward(self, x, edge_index, edge_weight):
        h = self.recurrent(x, edge_index, edge_weight).relu()
        h = self.linear(h)
        return h

model = TemporalGNN(dataset[0].x.shape[1])
```

В среднем модель EvolveGCN-O достигает схожих результатов со средним значением MSE 0.7524. В данном случае использование сети GRU или LSTM не влияет на прогнозы. Это вполне объяснимо, поскольку как прошлые значения количества посещений, содержащиеся в признаках узлов (EvolveGCN-H), так и связи между страницами (EvolveGCN-O) несут существенную информацию. В результате эта архитектура графовых нейронных сетей особенно подходит для задачи прогнозирования трафика.

Теперь, когда мы рассмотрели пример статического графа, давайте выясним, как обрабатывать динамические графы.

Прогнозирование случаев COVID-19

В этом разделе мы рассмотрим новую задачу – прогнозирование эпидемий. Воспользуемся **набором данных England Covid**, это динамический граф с темпоральной информацией, представленный Панагопулосом совместно с коллегами в 2021 году [3]. Несмотря на то что узлы статичны, связи между ними и веса ребер меняются с течением времени. Этот набор данных представляет собой количество зарегистрированных случаев COVID-19 в 129 регионах Англии (в соответствии со стандартом территориального деления NUTS 3) в период с 3 марта по 12 мая 2020 года. Данные были собраны с мобильных телефонов, на которых было установлено приложение Facebook и собирались данные о местоположении. Наша цель – предсказать количество случаев заболевания в каждом узле (регионе) за 1 день.

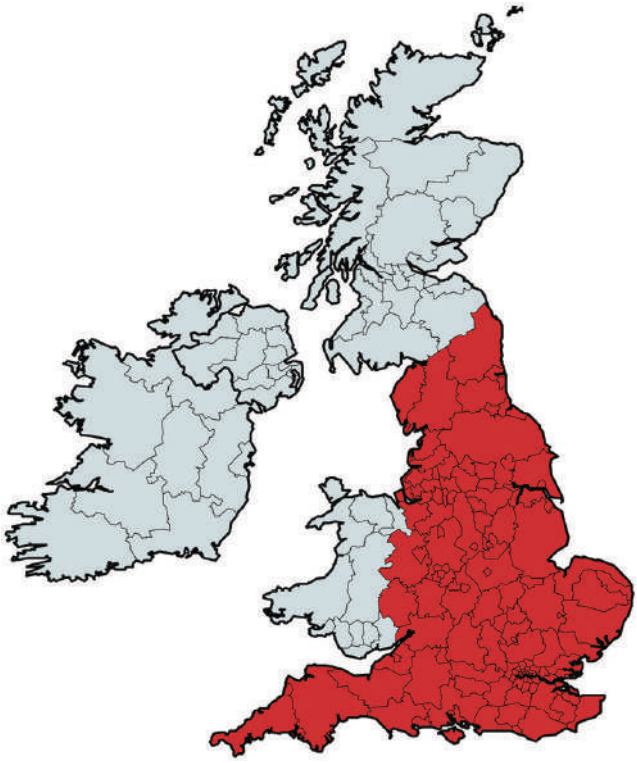


Рис. 13.8 ❖ Регионы Англии по стандарту NUTS 3 окрашены в красный цвет

Этот набор данных представляет Англию в виде графа $G = (V, E)$. В силу темпоральной природы этого набора данных граф состоит из нескольких графов, соответствующих каждому дню исследуемого периода $G^{(1)}, \ldots, G^{(T)}$. В этих графах признаки узлов соответствуют количеству случаев заболевания в каждый из прошедших d дней в данном регионе. Графы являются однонаправленными и взвешенными: вес $w_{v,u}^{(t)}$ ребра (v, u) представляет собой количество людей, переехавших из региона v в регион u в момент времени t. Эти графы также содержат самопетли, соответствующие перемещениям людей в пределах одного региона.

В этом разделе мы представим новую архитектуру GNN, разработанную для этой задачи, и покажем, как реализовать ее пошагово.

Знакомство с MPNN-LSTM

Как следует из названия, архитектура **MPNN-LSTM** основана на объединении нейросетей MPNN и LSTM. Как и набор данных England Covid, она также была представлена Панагопулосом совместно с коллегами в 2021 году [3].

Входные признаки узлов с соответствующими индексами ребер и весами подаются в GCN-слой. К выходным данным мы применяем слой батч-нормализации и дропаут. Этот процесс повторяется во второй раз применительно к результату первой MPNN. В итоге получаем матрицу эмбеддингов узлов $H^{(t)}$. Так мы создаем последовательность $H^{(1)}, \ldots, H^{(T)}$ представлений эмбеддингов узлов, применяя MPNN для каждого временного шага. Эта последовательность подается в двухслойную LSTM-сеть для захвата темпоральной информации из графов. Наконец, мы применяем линейное преобразование и функцию ReLU к результату, чтобы получить прогноз для $t + 1$.

На рис. 13.9 показано высокоуровневое представление архитектуры MPNN-LSTM.

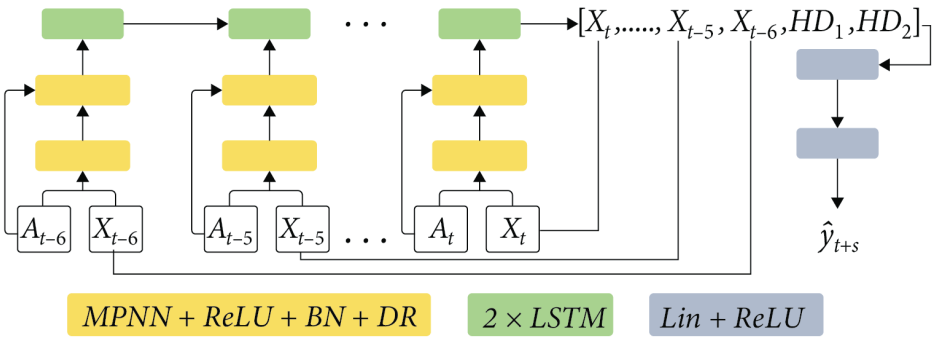


Рис. 13.9 ❖ Архитектура MPNN-LSTM

Авторы MPNN-LSTM отмечают, что эта модель не является лучшей по качеству для набора данных England Covid (таковой является MPNN с двухуров-

невой GNN). Однако это интересный подход, который может показать лучшие результаты в других сценариях. Они также утверждают, что он больше подходит для долгосрочного прогнозирования, например на 14 дней вперед, а не на один день, как в нашем варианте этого набора данных. Несмотря на эту проблему, мы воспользуемся последним вариантом для удобства, поскольку он не влияет на дизайн решения.

Реализация MPNN-LSTM

Во-первых, важно визуализировать количество случаев, которые мы хотим предсказать. Как и в предыдущем разделе, просуммируем 129 различных временных рядов, из которых состоит набор данных, вычислив их среднее значение и стандартное отклонение.

1. Импортируем функцию, которая загружает набор England Covid:

```
from torch_geometric_temporal.dataset import EnglandCovidDatasetLoader
```

2. Мы загружаем набор данных с 14 лагами, соответствующими количеству узловых признаков:

```
dataset = EnglandCovidDatasetLoader().get_dataset(lags=14)
```

3. Разбиваем данные на обучающую и тестовую выборки с учетом временной структуры (80 % - обучающая выборка, 20 % – тестовая выборка):

```
train_dataset, test_dataset = temporal_signal_split(dataset,
                                                    train_ratio=0.8)
```

4. Мы строим график, чтобы визуализировать среднее нормализованное количество зарегистрированных случаев COVID-19 (сообщения о них поступают примерно каждый день):

```
mean_cases = [snapshot.y.mean().item() for snapshot in dataset]
std_cases = [snapshot.y.std().item() for snapshot in dataset]
df = pd.DataFrame(mean_cases, columns=['mean'])
df['std'] = pd.DataFrame(std_cases, columns=['std'])

plt.figure(figsize=(10,5))
plt.plot(df['mean'], 'k-')
plt.grid(linestyle=':')
plt.fill_between(df.index,
                 df['mean'] - df['std'],
                 df['mean'] + df['std'],
                 color='r',
                 alpha=0.1)

plt.axvline(x=38, color='b', linestyle='--',
            label='Разбиение\побучение/тест')
```

```
plt.text(38, 1, 'Разбиение\nобучение/тест',
         rotation=-90, color='b')

plt.xlabel('Количество зарегистрированных случаев')
plt.ylabel('Среднее нормализованное количество случаев');
```

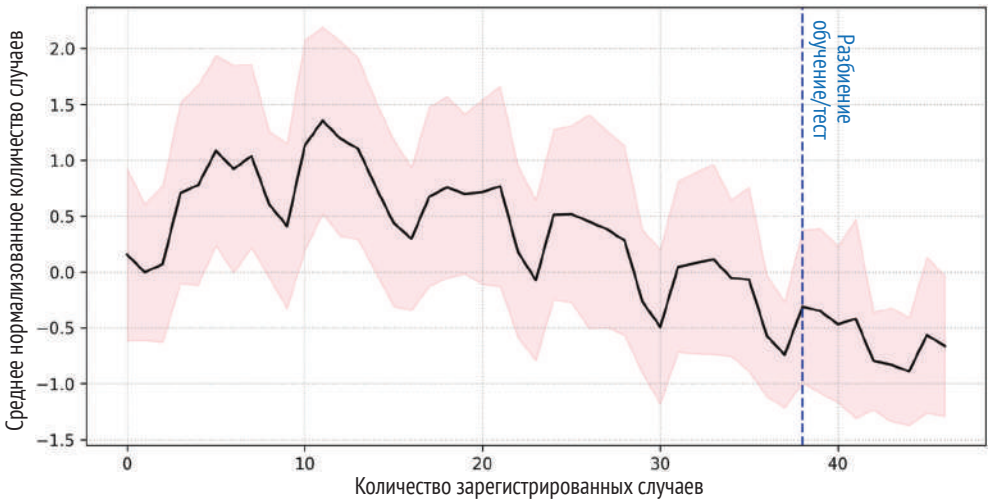


Рис. 13.10 ❖ Среднее нормализованное количество случаев (набор данных England Covid)

Этот график демонстрирует большую волатильность и небольшое количество снапшотов. Именно поэтому в данном примере мы использовали разбиение «обучение–тест» 80/20. Однако получить хорошее качество на таком небольшом наборе данных – непростая задача.

Теперь давайте реализуем архитектуру MPNN-LSTM.

1. Импортируем слой MPNNLSTM из PyTorch Geometric Temporal:

```
from torch_geometric_temporal.nn.recurrent import MPNNLSTM
```

2. Темпоральная GNN принимает на вход три параметра: количество входных признаков, размерность скрытых представлений и количество узлов. Мы объявляем три слоя: слой MPNN-LSTM, слой дропаута и линейный слой:

```
class TemporalGNN(torch.nn.Module):
    def __init__(self, dim_in, dim_h, num_nodes):
        super().__init__()
        self.recurrent = MPNNLSTM(dim_in, dim_h, num_nodes, 1, 0.5)
        self.dropout = torch.nn.Dropout(0.5)
        self.linear = torch.nn.Linear(2*dim_h + dim_in, 1)
```

3. Функция `forward()` учитывает веса ребер, что является важной информацией в данном наборе данных. Обратите внимание, что мы обра-

батываем динамический граф, поэтому на каждом временном шаге задается новый набор значений для `edge_index` и `edge_weight`. В отличие от оригинальной реализации MPNN-LSTM, описанной ранее, мы заменяем итоговую функцию ReLU на функцию `tanh`. Основная мотивация заключается в том, что tanh выводит значения в диапазоне от –1 до 1, а не в диапазоне от 0 до 1, это ближе к тому, что мы наблюдали в наборе данных:

```
    def forward(self, x, edge_index, edge_weight):
        h = self.recurrent(x, edge_index, edge_weight).relu()
        h = self.dropout(h)
        h = self.linear(h).tanh()
        return h
```

4. Мы создаем модель MPNN-LSTM с размерностью скрытых представлений 64 и печатаем информацию о ней, чтобы посмотреть слои, из которых она состоит:

```
model = TemporalGNN(dataset[0].x.shape[1],
                    64,
                    dataset[0].x.shape[0])
model
```

```
TemporalGNN(
  (recurrent): MPNNLSTM(
    (_convolution_1): GCNConv(14, 64)
    (_convolution_2): GCNConv(64, 64)
    (_batch_norm_1): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
    (_batch_norm_2): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_
running_stats=True)
    (_recurrent_1): LSTM(128, 64)
    (_recurrent_2): LSTM(64, 64)
  )
  (dropout): Dropout(p=0.5, inplace=False)
  (linear): Linear(in_features=142, out_features=1, bias=True)
)
```

Мы видим, что слой MPNN-LSTM содержит два GCN-слоя, два слоя батч-нормализации и два слоя LSTM (но без дропаута), что соответствует нашему предыдущему описанию.

5. Мы обучаем эту модель, задав 100 эпох, оптимизатор Adam и темп обучения 0.001:

```
set_seed()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
model.train()
```

```
# Обучение
for epoch in tqdm(range(100)):
```

```
        loss = 0
        for i, snapshot in enumerate(train_dataset):
            y_pred = model(snapshot.x,
                           snapshot.edge_index,
                           snapshot.edge_attr)
            loss = loss + torch.mean((y_pred-snapshot.y)**2)
        loss = loss / (i+1)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

6. Мы оцениваем качество обученной модели на тестовом наборе и получаем следующее значение функции потерь MSE:

```
# Оценка
model.eval()
loss = 0
for i, snapshot in enumerate(test_dataset):
    y_pred = model(snapshot.x,
                   snapshot.edge_index,
                   snapshot.edge_attr)
    mse = torch.mean((y_pred-snapshot.y)**2)
    loss += mse
loss = loss / (i+1)
print(f'MSE: {loss.item():.4f}')

MSE: 1.5005
```

Модель MPNN-LSTM дала значение функции потерь MSE 1.3722, что кажется довольно высоким показателем.

Мы не будем инвертировать процесс нормализации, который был применен к этому набору данных, поэтому визуализируем нормализованное количество случаев. Сначала построим график среднего нормализованного количества случаев, которые предсказала наша модель:

```
y_preds = [
    model(snapshot.x, snapshot.edge_index, snapshot.edge_attr)
    .squeeze()
    .detach()
    .numpy()
    .mean() for snapshot in test_dataset
]

plt.figure(figsize=(10,5), dpi=300)
plt.plot(df['mean'], 'k-')
plt.plot(range(38,48), y_preds, 'r-', label='Прогноз')
plt.grid(linestyle=':')
plt.fill_between(df.index, df['mean'] - df['std'],
                 df['mean'] + df['std'],
                 color='r',
                 alpha=0.1)
```

```
plt.axvline(x=38, color='b', linestyle='--',
            label='Разбиение\побучение/тест')

plt.text(38, 1.1, 'Разбиение\побучение/тест',
         rotation=-90, color='b')

plt.xlabel('Время (дни)')
plt.ylabel('Среднее нормализованное количество случаев');
```

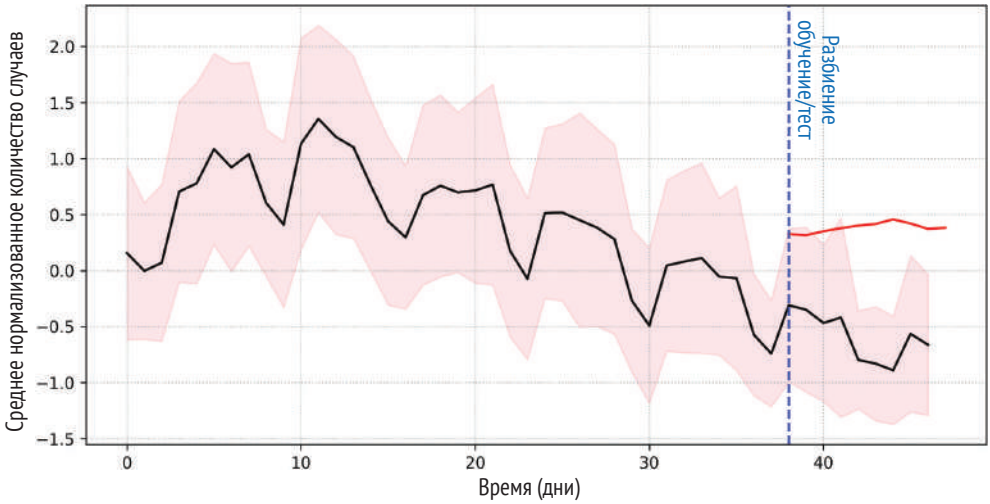


Рис. 13.11 ❖ Среднее нормализованное количество случаев (фактические значения – черная линия, прогнозы – красная линия)

Как и ожидалось, предсказанные значения не очень хорошо совпадают с фактическими. Вероятно, это связано с малым объемом данных: наша модель выучила среднее значение, которое минимизирует потери MSE, но не может подогнать кривую и понять ее периодичность.

Давайте посмотрим на диаграмму рассеяния, соответствующую первому снапшоту тестового набора:

```
import seaborn as sns

y_pred = (
    model(test_dataset[0].x,
          test_dataset[0].edge_index,
          test_dataset[0].edge_attr)
    .detach()
    .squeeze()
    .numpy()
)

plt.figure(figsize=(10,5), dpi=300)
sns.regplot(x=test_dataset[0].y.numpy(), y=y_pred);
```

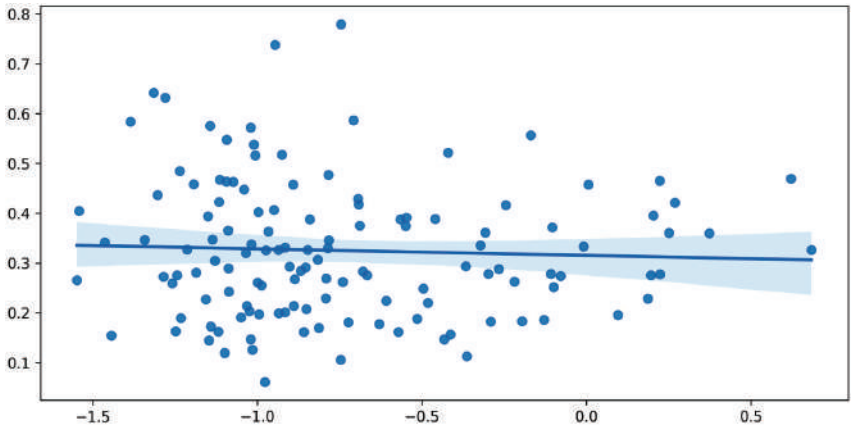


Рис. 13.12 ❖ Спрогнозированные и фактические значения для набора данных England Covid

Диаграмма рассеяния показывает слабую корреляцию. Мы видим, что прогнозы (ось y) в основном сосредоточены вокруг значения 0.35 с небольшим разбросом. Это не соответствует фактическим значениям, которые варьируют в диапазоне от –1.5 до 0.6. Согласно нашим экспериментам добавление второго линейного слоя не улучшило прогнозы MPNN-LSTM.

Чтобы помочь модели, можно реализовать несколько стратегий. Во-первых, большее количество данных могло бы значительно помочь, поскольку мы работали сейчас с небольшим набором данных. Кроме того, временной ряд содержит две интересные характеристики: тренды (постоянное увеличение и уменьшение с течением времени) и сезонность (предсказуемая закономерность). Мы можем добавить шаг предварительной обработки, чтобы удалить эти характеристики, которые добавляют шум в предсказываемый сигнал, с последующим восстановлением.

Помимо рекуррентных нейронных сетей, самовнимание – еще один популярный метод создания темпоральных GNN [4]. Механизм внимания может быть направлен только на темпоральную информацию или дополнительно учитывать еще и пространственные данные, что обычно делается с помощью графовой свертки. Наконец, темпоральные GNN можно также расширить на гетерогенные графы, описанные в предыдущей главе. К сожалению, такая комбинация потребует еще большего количества данных и в настоящее время является активной областью исследований.

Выводы

В этой главе был представлен новый тип графов, содержащий пространственно-темпоральную информацию. Этот темпоральный компонент будет полезен во многих задачах, в основном связанных с прогнозированием временных ря-

дов. Мы описали два типа графов, подходящих под это описание: статические графы, в которых характеристики изменяются со временем, и динамические графы, в которых могут меняться характеристики и топология. С обоими типами справляется PyTorch Geometric Temporal, расширение PyG, в котором реализованы нейронные сети для обучения на темпоральных графах.

Кроме того, мы рассмотрели две задачи с применением темпоральных GNN. Во-первых, реализовали архитектуру EvolveGCN, которая использует GRU или LSTM для обновления параметров GCN. Мы применили ее на примере прогнозирования веб-трафика – задачи, с которой мы сталкивались в главе 6 «Знакомство с графовыми сверточными нейронными сетями», – и получили отличные результаты на небольшом наборе данных. Во-вторых, мы использовали архитектуру MPNN-LSTM для прогнозирования эпидемий. Мы использовали набор данных England Covid – динамический граф с темпоральным сигналом, но его небольшой размер не позволил нам получить сопоставимые результаты.

В главе 14 «Интерпретация графовых нейронных сетей» мы сосредоточимся на том, как интерпретировать наши результаты. Помимо различных визуализаций, которые уже показывали, мы увидим, как применить к графовым нейронным сетям методы из области **интерпретируемого искусственного интеллекта** (eXplainable Artificial Intelligence, XAI). Эта область является ключевым компонентом для создания надежных систем ИИ и повышения эффективности машинного обучения. В этой главе мы представим методы объяснения post hoc и новые слои для построения моделей, которые по своей сути можно интерпретировать.

Дополнительное чтение

[1] A. Pareja et al., *EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs*. arXiv, 2019. DOI: 10.48550/ARXIV.1902.10191. Доступ по ссылке https://arxiv.org/abs/1902.10191.

[2] B. Rozemberczki et al., *PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models*, in Proceedings of the 30th ACM International Conference on Information and Knowledge Management, 2021, pp. 4564–4573. Доступ по ссылке https://arxiv.org/abs/2104.07788.

[3] G. Panagopoulos, G. Nikolentzos, and M. Vazirgiannis. *Transfer Graph Neural Networks for Pandemic Forecasting*. arXiv, 2020. DOI: 10.48550/ARXIV.2009.08388. Доступ по ссылке https://arxiv.org/abs/2009.08388.

[4] Guo, S., Lin, Y., Feng, N., Song, C., & Wan, H. (2019). Attention Based Spatial-Temporal Graph Convolutional Networks for Traffic Flow Forecasting. Proceedings of the AAAI Conference on Artificial Intelligence, 33 (01), 922–929. https://doi.org/10.1609/aaai.v33i01.3301922.

Глава 14

Интерпретация графовых нейронных сетей

Одним из наиболее распространенных критических замечаний в адрес графовых нейронных сетей является упрек в том, что их результаты трудно понять. К сожалению, графовые нейронные сети не застрахованы от этого ограничения: помимо объяснения того, какие признаки важны, необходимо учитывать соседние узлы и связи. В ответ на проблему трудности интерпретации было разработано множество методов (в виде **интерпретируемого ИИ** или **XAI**), позволяющих лучше понять причины, обусловившие тот или иной прогноз, или общее поведение модели. Некоторые из этих методов были адаптированы для применения к GNN, а другие используют преимущества структуры графа, чтобы предложить более точные объяснения.

В этой главе мы рассмотрим некоторые методы интерпретации, позволяющие понять причины, обусловившие тот или иной прогноз. Мы рассмотрим различные семейства методов и сосредоточимся на двух наиболее популярных – GNNExplainer и **методе интегрированных градиентов** (integrated gradients). Мы применим первый метод для задачи классификации графов, воспользовавшись набором данных `MUTAG`. Затем представим Python'овскую библиотеку Captum, которая предлагает множество методов интерпретации. Наконец, используя социальную сеть Twitch, мы применим интегрированные градиенты для объяснения результатов модели в задаче классификации узлов.

К концу этой главы вы сможете понять и реализовать несколько методов XAI применительно к GNN. Более конкретно – вы узнаете, как использовать GNNExplainer и библиотеку Captum (с интегрированными градиентами) для задач классификации графов и узлов.

В этой главе мы рассмотрим следующие основные темы:

- «Знакомство с методами интерпретации»,
- «Интерпретация GNN с помощью GNNExplainer»,
- «Интерпретация GNN с помощью Captum».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter14.

Знакомство с методами интерпретации

Интерпретация прогнозов GNN – это новая область, которая в значительной степени вдохновлена уже существующими методами XAI [1]. Мы разделяем ее на локальные объяснения (на уровне каждого прогноза) и глобальные объяснения (на уровне всей модели). Хотя понимание поведения модели GNN желательно, мы сосредоточимся на локальных объяснениях, которые более популярны и необходимы для интерпретации прогноза.

В этой главе мы проводим различие между моделями «белого ящика» и моделями «черного ящика». Модель называется моделью «белого ящика», если она понятна человеку по своей конструкции, как, например, дерево решений. С другой стороны, она является «черным ящиком», если ее прогнозы можно понять, только используя методы интерпретации. Так обычно происходит с нейронными сетями: их веса и смещения не дают четких правил, как дерево решений, но их результаты можно объяснить косвенным образом.

Существует четыре основные категории локальных методов объяснения.

- **Методы, основанные на градиентах** (gradient-based methods), анализируют градиенты выходных данных для вычисления оценок атрибуции, например **метод интегральных градиентов**.
- **Методы, основанные на пертурбациях** (perturbation-based methods), маскируют или изменяют входные признаки для измерения изменений в выходных данных, например **GNNExplainer**.
- **Методы декомпозиции** раскладывают прогнозы модели на несколько компонент, чтобы оценить их, например **графовая нейронная сеть с послойным распространением релевантности** (graph neural network layer-wise relevance propagation – GNN-LRP).
- **Суррогатные методы** (surrogate methods) используют простую и интерпретируемую модель для аппроксимации прогноза исходной модели в определенной области (например, GraphLIME).

Эти методы дополняют друг друга: они иногда дают противоположные оценки вкладов ребер и признаков, их можно использовать, чтобы в дальнейшем уточнить интерпретацию прогноза. Методы интерпретации традиционно оцениваются с помощью следующих метрик.

- **Верность** (fidelity), которая сравнивает вероятности прогнозов y_i для исходного графа G_i и модифицированного графа $\hat{G}_i$. В модифицированном графе сохраняются только наиболее важные признаки (признаки

узлов, ребер) графа на основе объяснения $\hat{y}_i$. Другими словами, верность измеряет степень того, насколько выбранные важные признаки достаточны для получения правильного прогноза. Формально она определяется следующим образом:

$$Fidelity = \frac{1}{N}\sum_{i=1}^{N}(f(G_i)_{y_i} - f(\hat{G}_i)_{y_i}).$$

- **Разреженность** (sparsity), которая измеряет долю важных признаков (признаков узлов, ребер). Слишком длинные объяснения сложнее понять, поэтому разреженность является желательной характеристикой. Она вычисляется следующим образом:

$$Sparsity = \frac{1}{N}\sum_{i=1}^{N}\left(1 - \frac{|m_i|}{|M_i|}\right),$$

где $|m_i|$ – это количество важных входных признаков и $|M_i|$ – это общее количество признаков.

Методы интерпретации часто оцениваются на синтетических наборах данных типа `BA-Shapes`, `BA-Community`, `Tree-Cycles` и `Tree-Grid` [2]. Эти наборы данных были сгенерированы с помощью алгоритмов генерации графов для создания определенных паттернов. Мы не будем использовать их в этой главе, но они представляют собой интересные альтернативные наборы данных, которые можно использовать для иллюстрации того или иного метода интерпретации.

В следующих разделах мы опишем метод на основе градиентов (интегрированные градиенты) и метод на основе пертурбаций (GNNExplainer).

Интерпретация графовых нейронных сетей с помощью GNNExplainer

В этом разделе мы познакомимся с нашим первым методом XAI – GNNExplainer. Воспользуемся им для понимания прогнозов, полученных с помощью модели GIN для набора данных `MUTAG`.

Знакомство с GNNExplainer

Представленный в 2019 году Ином и др. [2], метод GNNExplainer представляет собой архитектуру GNN, которая предназначена для интерпретации прогнозов, полученных с помощью другой модели GNN. При работе с табличными данными мы хотим знать, какие признаки вносят наибольший вклад в про-

гноз. Однако в случае с графовыми данными этого недостаточно: нам также нужно знать, какие узлы являются наиболее влиятельными. GNNExplainer генерирует прогнозы с учетом обоих вышеупомянутых аспектов, выделяя подграф G_S и подмножество признаков узлов X_S. На рис. 14.1 показано объяснение, сгенерированное GNNExplainer для конкретного узла.

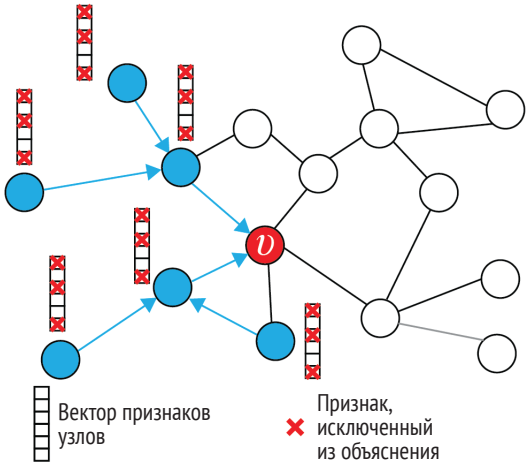


Рис. 14.1 ❖ Интерпретация для метки узла v с помощью подграфа G_S (выделен синим) и подмножества неисключенных признаков узлов X_S

Для прогнозирования G_S и X_S GNNExplainer использует маску ребер (для скрытия связей) и маску признаков (для скрытия признаков узлов). Если связь или признак важны, то их удаление должно кардинально изменить прогноз. С другой стороны, если прогноз не меняется, значит, эта информация была избыточной или просто была нерелевантной (т. е. не связана с зависимой переменной). Этот принцип лежит в основе методов, основанных на пертурбациях, типа GNNExplainer.

На практике мы должны тщательно подбирать функцию потерь, чтобы найти наилучшую из возможных масок. GNNExplainer измеряет взаимную зависимость между распределением спрогнозированных меток и (G_S, X_S), также называемую **взаимной информацией** (mutual information – MI). Наша цель – максимизировать MI, что эквивалентно минимизации условной кросс-энтропии. GNNExplainer обучается находить переменные G_S и X_S, которые максимизируют вероятность прогноза $\hat{y}_i$.

Помимо этой оптимизационной схемы, GNNExplainer выучивает бинарную маску признаков и применяет несколько методов регуляризации. Наиболее важным моментом является минимизация размера объяснения (разреженность). Размер объяснения вычисляется как сумма всех элементов параметров маски и добавляется к функции потерь. Это позволяет создавать более удобные и лаконичные объяснения, которые легче интерпретировать.

GNNExplainer может применяться к большинству архитектур GNN и различным задачам, таким как классификация узлов, прогнозирование связей или классификация графов. Кроме того, он может генерировать объяснения метки класса или всего графа. Для классификации графов модель рассматривает не одну, а совокупность матриц смежности для всех узлов графа. В следующем разделе мы применим эту модель для объяснения результатов классификации графов.

Реализация GNNExplainer

В этом примере мы исследуем набор данных MUTAG [3]. Каждый из 188 графов в этом наборе данных представляет химическое соединение, где узлы – это атомы (семь возможных атомов), а ребра – химические связи (четыре возможные связи). Признаки узлов и ребер представляют собой one-hot-кодировки типов атомов и ребер соответственно. Задача состоит в том, чтобы разделить каждое соединение на два класса в соответствии с их мутагенным действием на бактерию *Salmonella typhimurium*.

Для классификации белков мы будем использовать модель GIN, представленную в главе 9. В главе 9 мы визуализировали правильные и неправильные классификации, выполненные моделью. Однако мы не смогли объяснить прогнозы, сделанные GNN. На этот раз мы воспользуемся GNNExplainer, чтобы понять, какие признаки подграфа и узла наиболее важны для объяснения классификации. В этом примере мы проигнорируем признаки ребер в целях простоты использования.

1. Импортируем необходимые классы из PyTorch и PyTorch Geometric:

```
import matplotlib.pyplot as plt

import torch.nn.functional as F
from torch.nn import Linear, Sequential, BatchNorm1d, ReLU, Dropout

from torch_geometric.datasets import TUDataset
from torch_geometric.loader import DataLoader
from torch_geometric.nn import GINConv, global_add_pool, GNNExplainer
```

2. Нам нужно обеспечить воспроизводимость результатов:

```
set_seed()
```

3. Загружаем набор данных MUTAG и перемешиваем данные:

```
dataset = TUDataset(root='data/TUDataset', name='MUTAG').shuffle()
```

4. Мы создаем обучающий, проверочный и тестовый наборы:

```
# создаем обучающий, проверочный и тестовый наборы
train_dataset = dataset[:int(len(dataset)*0.8)]
val_dataset   = dataset[int(len(dataset)*0.8):int(len(dataset)*0.9)]
test_dataset  = dataset[int(len(dataset)*0.9):]
```

5. Создаем загрузчики данных для реализации мини-батчей:

```
# создаем мини-батчи
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader   = DataLoader(val_dataset, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_dataset, batch_size=64, shuffle=True)
```

6. Создаем модель GIN со скрытой размерностью 32, используя программный код из главы 9:

```
class GIN(torch.nn.Module):
    """GIN"""
    def __init__(self, dim_h):
        super(GIN, self).__init__()
        self.conv1 = GINConv(
            Sequential(Linear(dataset.num_node_features, dim_h),
                       BatchNorm1d(dim_h), ReLU(),
                       Linear(dim_h, dim_h), ReLU()))
        self.conv2 = GINConv(
            Sequential(Linear(dim_h, dim_h), BatchNorm1d(dim_h), ReLU(),
                       Linear(dim_h, dim_h), ReLU()))
        self.conv3 = GINConv(
            Sequential(Linear(dim_h, dim_h), BatchNorm1d(dim_h), ReLU(),
                       Linear(dim_h, dim_h), ReLU()))
        self.lin1 = Linear(dim_h*3, dim_h*3)
        self.lin2 = Linear(dim_h*3, dataset.num_classes)

    def forward(self, x, edge_index, batch):
        h1 = self.conv1(x, edge_index)
        h2 = self.conv2(h1, edge_index)
        h3 = self.conv3(h2, edge_index)

        h1 = global_add_pool(h1, batch)
        h2 = global_add_pool(h2, batch)
        h3 = global_add_pool(h3, batch)

        h = torch.cat((h1, h2, h3), dim=1)

        h = self.lin1(h)
        h = h.relu()
        h = F.dropout(h, p=0.5, training=self.training)
        h = self.lin2(h)

        return F.log_softmax(h, dim=1)

model = GIN(dim_h=32)
```

7. Мы обучаем эту модель, задав 100 эпох, и оцениваем ее качество, используя программный код из главы 9:

```
@torch.no_grad()
def test(model, loader):
    criterion = torch.nn.CrossEntropyLoss()
    model.eval()
```

```
    loss = 0
    acc = 0

    for data in loader:
        out = model(data.x, data.edge_index, data.batch)
        loss += criterion(out, data.y) / len(loader)
        acc += accuracy(out.argmax(dim=1), data.y) / len(loader)

    return loss, acc

def accuracy(pred_y, y):
    return ((pred_y == y).sum() / len(y)).item()

criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
epochs = 200

model.train()
for epoch in range(epochs+1):
    total_loss = 0
    acc = 0
    val_loss = 0
    val_acc = 0

    # Обучаем на батчах
    for data in train_loader:
        optimizer.zero_grad()
        out = model(data.x, data.edge_index, data.batch)
        loss = criterion(out, data.y)
        total_loss += loss / len(train_loader)
        acc += accuracy(out.argmax(dim=1), data.y) / len(train_loader)
        loss.backward()
        optimizer.step()

        # Проверка
        val_loss, val_acc = test(model, val_loader)

    # Печатаем метрики через каждые 20 эпох
    if epoch % 20 == 0:
        print(f'Эпоха {epoch:>3}:\n| Функция потерь на обуч. наборе: '
              f'{total_loss:.2f} | Правильность на обуч. наборе: '
              f'{acc*100:>5.2f}% \n| Функция потерь на провер. наборе: '
              f'{val_loss:.2f} | Правильность на провер. наборе: '
              f'{val_acc*100:.2f}%')

test_loss, test_acc = test(model, test_loader)
print(f'\nФункция потерь на тестовом наборе: {test_loss:.2f} | '
      f'Правильность на тестовом наборе: {test_acc*100:.2f}%')
```

```
Эпоха   0:
| Функция потерь на обуч. наборе: 1.93 | Правильность на обуч. наборе: 65.39%
| Функция потерь на провер. наборе: 0.61 | Правильность на провер. наборе: 68.42%
Эпоха  20:
| Функция потерь на обуч. наборе: 0.46 | Правильность на обуч. наборе: 79.88%
```

```
| Функция потерь на провер. наборе: 0.35 | Правильность на провер. наборе: 89.47%
Эпоха  40:
| Функция потерь на обуч. наборе: 0.39 | Правильность на обуч. наборе: 82.48%
| Функция потерь на провер. наборе: 0.33 | Правильность на провер. наборе: 94.74%
Эпоха  60:
| Функция потерь на обуч. наборе: 0.33 | Правильность на обуч. наборе: 81.44%
| Функция потерь на провер. наборе: 0.35 | Правильность на провер. наборе: 89.47%
Эпоха  80:
| Функция потерь на обуч. наборе: 0.39 | Правильность на обуч. наборе: 77.89%
| Функция потерь на провер. наборе: 0.54 | Правильность на провер. наборе: 68.42%
Эпоха 100:
| Функция потерь на обуч. наборе: 0.37 | Правильность на обуч. наборе: 81.53%
| Функция потерь на провер. наборе: 0.21 | Правильность на провер. наборе: 94.74%
Эпоха 120:
| Функция потерь на обуч. наборе: 0.35 | Правильность на обуч. наборе: 85.13%
| Функция потерь на провер. наборе: 0.47 | Правильность на провер. наборе: 73.68%
Эпоха 140:
| Функция потерь на обуч. наборе: 0.24 | Правильность на обуч. наборе: 86.13%
| Функция потерь на провер. наборе: 0.18 | Правильность на провер. наборе: 94.74%
Эпоха 160:
| Функция потерь на обуч. наборе: 0.21 | Правильность на обуч. наборе: 91.67%
| Функция потерь на провер. наборе: 0.21 | Правильность на провер. наборе: 94.74%
Эпоха 180:
| Функция потерь на обуч. наборе: 0.30 | Правильность на обуч. наборе: 87.12%
| Функция потерь на провер. наборе: 0.11 | Правильность на провер. наборе: 94.74%
Эпоха 200:
| Функция потерь на обуч. наборе: 0.15 | Правильность на обуч. наборе: 93.75%
| Функция потерь на провер. наборе: 0.17 | Правильность на провер. наборе: 94.74%

Функция потерь на тестовом наборе: 0.50 | Правильность на тестовом наборе: 84.21%
```

8. Наша модель GIN обучена и достигла высокой правильности (84.21 %). Теперь давайте создадим модель GNNExplainer, используя класс `GNNExplainer` библиотеки PyTorch Geometric. Мы обучим ее за 100 эпох:

```
explainer = GNNExplainer(model, epochs=100, num_hops=1)
```

9. Класс `GNNExplainer` можно использовать для объяснения предсказания, сделанного для узла (`.explain_node()`) или всего графа (`.explain_graph()`). В данном случае мы будем использовать его для последнего графа тестового набора:

```
explainer = GNNExplainer(model, epochs=100, num_hops=1)
data = dataset[-1]
feature_mask, edge_mask = explainer.explain_graph(data.x, data.edge_index)
```

10. На последнем этапе были получены маски признаков и ребер. Давайте распечатаем маску признаков, чтобы увидеть наиболее важные значения:

```
feature_mask

tensor([0.6788, 0.6915, 0.6694, 0.2613, 0.2655, 0.2748, 0.2574])
```

Значения нормированы между 0 (менее важно) и 1 (более важно). Эти семь значений соответствуют семи атомам, которые мы находим в наборе данных в следующем порядке: углерод (C), азот (N), кислород (O), фтор (F), йод (I), хлор (Cl) и бром (Br). Наиболее полезным является второй признак, представляющая азот (N), а наименее важным – последний, седьмой признак, представляющий бром (Br).

11. Вместо того чтобы распечатывать маску ребер, мы можем визуализировать ее на графе с помощью метода `.visualize_graph()`. Степень непрозрачности стрелок отражает важность каждого соединения:

```
fig = plt.figure(dpi=200)
ax, G = explainer.visualize_subgraph(
    -1, data.edge_index, edge_mask, y=data.y
)
ax.axis('off')
plt.show()
```

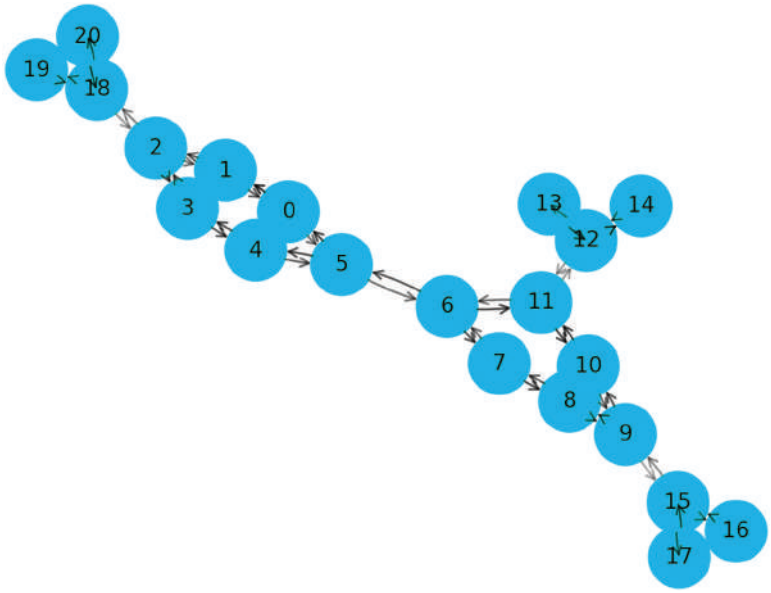


Рис. 14.2 ❖ Графическое представление химического соединения: непрозрачность ребер отражает важность каждой связи

Последний график показывает связи, которые внесли наибольший вклад в прогноз. В данном случае модель GIN правильно классифицировала граф. Видно, что здесь наиболее релевантными являются связи между узлами 15, 16, 17, 18, 19 и 20. Выделенные связи сыграли решающую роль в классификации этого химического соединения. Мы можем узнать о них больше, распечатав `data.edge_attr`, чтобы получить метку, связанную с этими химическими связями (ароматические, одинарные, двойные или тройные).

Распечатав файл `data.x`, мы также можем посмотреть на узлы 15, 16, 17, 18, 19 и 20, чтобы получить дополнительную информацию. Ее следует сообщить людям, обладающим необходимыми знаниями в данной области, чтобы получить отзывы о нашей модели.

GNNExplainer не дает точных правил процесса принятия решений, но дает представление о том, на что ориентировалась модель GNN при составлении своего прогноза. Человеческий опыт все еще необходим для того, чтобы убедиться, что эти идеи последовательны и соответствуют традиционным знаниям о домене.

В следующем разделе мы воспользуемся библиотекой Captum для объяснения классификации узлов в социальной сети.

Интерпретация графовых нейронных сетей с помощью Captum

В этом разделе мы сначала познакомимся с библиотекой Captum и методом интегрированных градиентов, применяемым к графовым данным. Затем реализуем его с помощью модели PyTorch Geometric, взяв в качестве примера социальную сеть Twitch.

Знакомство с Captum и методом интегрированных градиентов

Captum (captum.ai) – это библиотека Python, реализующая множество современных алгоритмов интерпретации моделей PyTorch. Эта библиотека не ограничивается только GNN, она также может применяться к тексту, изображениям, табличным данным и т. д. Она весьма полезна, поскольку позволяет пользователям быстро тестировать различные техники и сравнивать разные объяснения одного и того же прогноза. Кроме того, в Captum реализованы такие популярные алгоритмы, как LIME и Gradient SHAP, для оценки вклада на уровне сети, слоя и нейрона.

В этом разделе мы воспользуемся этой библиотекой, чтобы применить графовый вариант метода интегрированных градиентов [4]. Этот метод присваивает оценку вклада каждому входному признаку. Captum использует градиенты относительно входных данных модели. Конкретно он использует вход x и базовый вход x' (все связи имеют нулевой вес). Он вычисляет градиенты во всех точках вдоль пути между x и x' и аккумулирует их.

Формально интегрированный градиент по i-й размерности для входа x определяется следующим образом:

$$IntegratedGrads_i(x) ::= (x_i - x_i') \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha \times (x - x'))}{\partial x_i} d\alpha.$$

На практике вместо непосредственного вычисления этого интеграла мы аппроксимируем его дискретной суммой.

Интегрированные градиенты не зависят от модели и опираются на две аксиомы:

- **чувствительность** (sensitivity): каждый вход, вносящий вклад в прогноз, должен получить ненулевую оценку;
- **инвариантность реализации** (implementation invariance): две нейронные сети, выходы которых одинаковы для всех входов (такие сети называются функционально эквивалентными), должны иметь одинаковые вклады признаков.

Графовый вариант метода интегрированных градиентов вместо признаков рассматривает узлы и ребра. В результате вы можете увидеть, что результат отличается от GNNExplainer, который учитывает признаки узлов и ребра. Вот почему эти два подхода могут дополнять друг друга.

Давайте теперь реализуем этот метод и визуализируем результаты.

Реализация метода интегрированных градиентов

Мы применим интегрированные градиенты на новом наборе данных: наборе данных социальной сети Twitch (английская версия) [5]. Он представляет собой граф «пользователь–пользователь», где узлы соответствуют стримерам Twitch, а связи – взаимной дружбе. 128 характеристик узлов представляют собой такую информацию, как характеристики стриминга, местоположение, любимые игры и т. д. Задача состоит в том, чтобы определить, использует ли стример ненормативный язык (бинарная классификация).

Для этой задачи мы реализуем простую двухслойную графовую сверточную нейронную сеть (GCN) с помощью PyTorch Geometric. Затем преобразуем нашу модель в формат Captum, чтобы воспользоваться алгоритмом интегрированных градиентов, и интерпретируем результаты.

1. Нам нужно установить библиотеку captum:

```
! pip install captum
```

2. Импортируем необходимые библиотеки:

```
from captum.attr import IntegratedGradients

import torch_geometric.transforms as T
from torch_geometric.datasets import Twitch
from torch_geometric.nn import Explainer, GCNConv, to_captum
```

3. Нам нужно обеспечить воспроизводимость результатов:

```
set_seed()
```

4. Загружаем набор данных социальной сети Twitch (английская версия):

```
dataset = Twitch('.', name="EN")
data = dataset[0]
```

5. На этот раз воспользуемся простой двухслойной графовой сверточной нейронной сетью с дропаутом:

```
class GCN(torch.nn.Module):
    def __init__(self, dim_h):
        super().__init__()
        self.conv1 = GCNConv(dataset.num_features, dim_h)
        self.conv2 = GCNConv(dim_h, dataset.num_classes)

    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index).relu()
        h = F.dropout(h, p=0.5, training=self.training)
        h = self.conv2(h, edge_index)
        return F.log_softmax(h, dim=1)
```

6. Мы попробуем обучить модель на GPU, если будет такая возможность, с помощью оптимизатора Adam:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCN(64).to(device)
data = data.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=5e-4)
```

7. Мы обучаем модель в течение 200 эпох, используя функцию потерь отрицательного логарифмического правдоподобия:

```
for epoch in range(200):
    model.train()
    optimizer.zero_grad()
    log_logits = model(data.x, data.edge_index)
    loss = F.nll_loss(log_logits, data.y)
    loss.backward()
    optimizer.step()
```

8. Протестируем качество модели. Обратите внимание, что мы не указали никакого тестового набора, поэтому в данном случае мы будем оценивать правильность модели GCN на обучающем наборе:

```
def accuracy(pred_y, y):
    return ((pred_y == y).sum() / len(y)).item()

@torch.no_grad()
def test(model, data):
    model.eval()
    out = model(data.x, data.edge_index)
    acc = accuracy(out.argmax(dim=1), data.y)
    return acc

acc = test(model, data)
```

```
print(f'Правильность: {acc*100:.2f}%')
```

```
Правильность: 68.80%
```

Правильность модели составила 68.80 %, что относительно мало, учитывая, что качество модели оценивалось на обучающем наборе.

9. Теперь можно приступить к реализации выбранного нами метода интерпретации – метода интегрированных градиентов. Сначала мы должны указать узел, который хотим объяснить (узел 0 в этом примере), и преобразовать модель PyTorch Geometric в формат Captum. Здесь мы также указываем, что хотим использовать маску признаков и ребер с помощью значения параметра `mask_type='node_and_edge'`:

```
node_idx = 0
captum_model = to_captum(model,
                         mask_type='node_and_edge',
                         output_idx=node_idx)
```

10. Давайте создадим объект `IntegratedGradients` с помощью Captum. В качестве входных данных передадим ему результат предыдущего шага:

```
ig = IntegratedGradients(captum_model)
```

11. У нас уже есть маска узлов, которую нужно передать Captum (`data.x`), но нам нужно создать тензор для маски ребер. В этом примере мы хотим учитывать каждое ребро в графе, поэтому инициализируем тензор ребер размера `data.num_edges`:

```
edge_mask = torch.ones(data.num_edges,
                       requires_grad=True,
                       device=device)
```

12. Метод `.attribute()` принимает определенный формат входных данных для масок узлов и ребер (отсюда и использование `.unsqueeze(0)` для переформатирования этих тензоров). Параметру `target` соответствует класс интересующего нас узла. Наконец, для параметра `additional_forward_args` мы указываем матрицу смежности (`data.edge_index`):

```
attr_node, attr_edge = ig.attribute(
    (data.x.unsqueeze(0), edge_mask.unsqueeze(0)),
    target=int(data.y[node_idx]),
    additional_forward_args=(data.edge_index),
    internal_batch_size=1)
```

13. Мы масштабируем оценки внимания в диапазон между 0 и 1:

```
attr_node = attr_node.squeeze(0).abs().sum(dim=1)
attr_node /= attr_node.max()
attr_edge = attr_edge.squeeze(0).abs()
attr_edge /= attr_edge.max()
```

14. С помощью класса `Explainer` в PyTorch Geometric визуализируем графическое представление этих атрибуций:

```
fig = plt.figure(dpi=200)
explainer = Explainer(model)
ax, G = explainer.visualize_subgraph(
    node_idx, data.edge_index, attr_edge,
    node_alpha=attr_node, y=data.y
)
ax.axis('off')
plt.show()
```

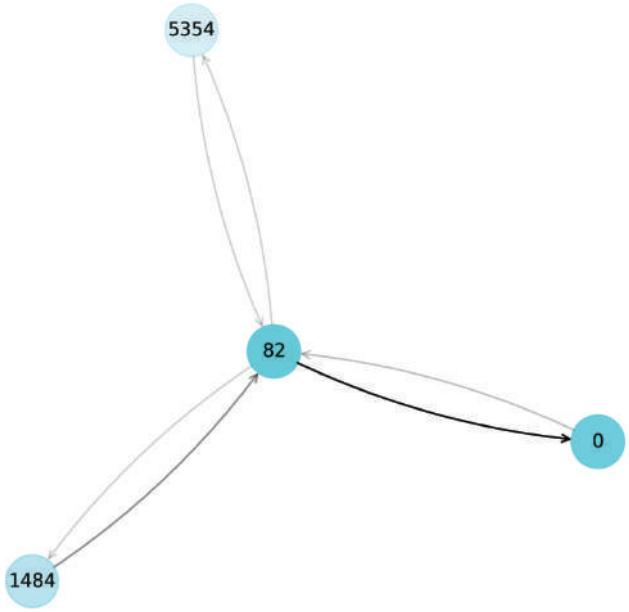


Рис. 14.3 ❖ Пояснения к классификации узла 0 с оценками вкладов (атрибуциями) ребер и узлов, представленными с помощью разной степени непрозрачности

Подграф узла **0** состоит из синих узлов одного и того же класса. Мы видим, что узел **82** является самым важным узлом (помимо узла **0**), а связь между этими двумя узлами – это самое важное ребро. Этому есть простое объяснение: у нас есть группа из четырех стримеров, использующих один и тот же язык. Взаимная дружба между узлами **0** и **82** – хороший аргумент в пользу этого прогноза.

Теперь давайте посмотрим на другой график, изображенный на рис. 14.4, – объяснение классификации узла **101**:

```
node_idx = 101
captum_model = to_captum(model,
                         mask_type='node_and_edge',
```

```
                          output_idx=node_idx)
ig = IntegratedGradients(captum_model)
edge_mask = torch.ones(data.num_edges,
                       requires_grad=True,
                       device=device)

attr_node, attr_edge = ig.attribute(
    (data.x.unsqueeze(0), edge_mask.unsqueeze(0)),
    target=int(data.y[node_idx]),
    additional_forward_args=(data.edge_index),
    internal_batch_size=1)

attr_node = attr_node.squeeze(0).abs().sum(dim=1)
attr_node /= attr_node.max()
attr_edge = attr_edge.squeeze(0).abs()
attr_edge /= attr_edge.max()

fig = plt.figure(dpi=200)
explainer = Explainer(model)
ax, G = explainer.visualize_subgraph(
    node_idx, data.edge_index,
    attr_edge, node_alpha=attr_node, y=data.y
)
ax.axis('off')
plt.show()
```

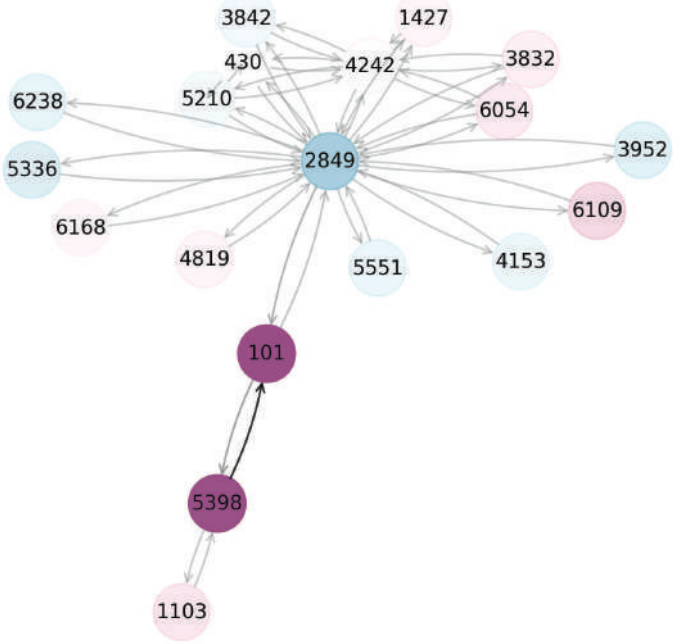


Рис. 14.4 ❖ Пояснения к классификации узла 101 с оценками вкладов (атрибуциями) ребер и узлов, представленными с помощью разной степени непрозрачности

В данном случае наш целевой узел связан с соседями, которые принадлежат к разным классам (узлы **5398** и **2849**). Метод интегрированных градиентов придает большую важность узлу, который принадлежит к тому же классу, что и узел **101**. Мы также видим, что связь между узлами **101** и **5398** вносит наибольший вклад в эту классификацию. Этот подграф содержит больше информации; видно, что даже соседи второго уровня вносят небольшой вклад.

Однако эти объяснения не следует считать серебряной пулей. Интерпретируемость в области ИИ – это обширная тема, в которой часто участвуют люди с разным опытом. Поэтому особенно важно делиться результатами и регулярно получать обратную связь. Информация о важности ребер, узлов и признаков очень важна, но она должна быть только началом обсуждения. Эксперты из других областей могут использовать или уточнить эти объяснения и даже найти проблемы, которые могут привести к изменению архитектуры.

Выводы

В этой главе мы рассмотрели область XAI с точки зрения применения к GNN. Интерпретируемость является ключевым компонентом во многих областях и может помочь нам построить модели более высокого качества. Мы рассмотрели различные методы получения локальных объяснений и сосредоточились на GNNExplainer (методе, основанном на пертурбациях) и методе интегрированных градиентов (методе, основанном на градиентах). Мы применили их к двум разным наборам данных с помощью PyTorch Geometric и Captum, чтобы получить объяснения для классификации графов и узлов. Наконец, мы визуализировали и обсудили результаты применения этих методов.

В главе 15 «Прогнозирование трафика с помощью A3T-GCN» мы вновь обратимся к темпоральным GNN для прогнозирования дорожного трафика. В этой практической задаче мы увидим, как преобразовать дороги в графы и применить новейшую архитектуру GNN для точного краткосрочного прогнозирования трафика.

Дополнительное чтение

[1] H. Yuan, H. Yu, S. Gui, and S. Ji. *Explainability in Graph Neural Networks: A Taxonomic Survey*. arXiv, 2020. DOI: 10.48550/ARXIV.2012.15445. Доступ по ссылке https://arxiv.org/abs/2012.15445.

[2] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. *GNNExplainer: Generating Explanations for Graph Neural Networks*. arXiv, 2019. DOI: 10.48550/ARXIV.1903.03894. Доступ по ссылке https://arxiv.org/abs/1903.03894.

[3] Debnath, A. K., Lopez de Compadre, R. L., Debnath, G., Shusterman, A. J., and Hansch, C. (1991). *Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity*. DOI: 10.1021/ jm00106a046. *Journal of Medicinal Chemistry*, 34 (2), 786–797. Доступ по ссылке https://doi.org/10.1021/jm00106a046.

[4] M. Sundararajan, A. Taly, and Q. Yan. *Axiomatic Attribution for Deep Networks*. arXiv, 2017. DOI: 10.48550/ARXIV.1703.01365. Доступ по ссылке https://arxiv.org/abs/1703.01365.

[5] B. Rozemberczki, C. Allen, and R. Sarkar. *Multi-Scale Attributed Node Embedding*. *arXiv*, *2019*. DOI: 10.48550/ARXIV.1909.13021. Доступ по ссылке https://arxiv.org/pdf/1909.13021.pdf.

ЧАСТЬ IV

ЗАДАЧИ

В этой четвертой и заключительной части книги мы углубимся в решение реальных задач. Мы сосредоточимся на аспектах, которые ранее не рассматривались в предыдущих главах, типа разведочного анализа данных и предварительной обработки данных. Постараемся дать исчерпывающий обзор конвейера машинного обучения, начиная с исходных данных и заканчивая анализом результатов моделирования. Мы также подчеркнем сильные стороны и ограничения обсуждаемых методов.

Проекты, представленные в этом разделе, разработаны таким образом, чтобы их можно было адаптировать и настраивать, что позволит читателям с легкостью применять их для других наборов данных и задач. Это делает их идеальным ресурсом для читателей, которые хотят создать портфолио проектов и продемонстрировать свои работы на GitHub.

К концу этой части вы узнаете, как реализовать GNN для прогнозирования дорожного трафика, обнаружения аномалий и создания рекомендательных систем. Эти проекты были выбраны для того, чтобы продемонстрировать универсальность и потенциал GNN в решении реальных задач. Знания и навыки, полученные в ходе работы над этими проектами, подготовят читателей к разработке собственных приложений и позволят внести им свой вклад в обучение на графах.

Эта часть включает в себя следующие главы:

- глава 15 «Прогнозирование трафика с помощью A3T-GCN»;
- глава 16 «Построение рекомендательной системы с помощью LightGCN»;
- глава 17 «Обнаружение аномалий с помощью гетерогенных графовых нейронных сетей»;
- глава 18 «Раскрытие потенциала графовых нейронных сетей в реальных задачах».

Глава 15

Прогнозирование трафика с помощью A3T-GCN

В главе 13 мы познакомились с T-GNN, но не разобрали подробно их основное применение – **прогнозирование дорожного трафика** (traffic forecasting). В последние годы все большую популярность приобретает концепция «умных городов». Эта идея относится к городам, в которых данные используются для управления и улучшения операций и услуг. В этом контексте одним из основных источников привлекательности является создание интеллектуальных транспортных систем. Точные прогнозы дорожного трафика могут помочь менеджерам, занимающимся вопросами транспорта, оптимизировать работу светофоров, планировать инфраструктуру и снижать количество заторов. Однако прогнозирование дорожного трафика – сложная задача из-за сложных пространственных и темпоральных зависимостей.

В этой главе мы применим T-GNN к конкретной задаче прогнозирования дорожного трафика. Сначала исследуем и предварительно обработаем новый набор данных, чтобы создать темпоральный граф из исходных CSV-файлов. Затем применим новый тип T-GNN для прогнозирования будущей скорости движения. Наконец, мы визуализируем и сравним результаты с базовыми решениями, чтобы убедиться в актуальности нашей архитектуры.

К концу этой главы вы узнаете, как создать набор данных о темпоральном графе на основе табличных данных. В частности, мы увидим, как создать взвешенную матрицу смежности, которая обеспечит нас весами ребер. Наконец, вы узнаете, как применить T-GNN к задаче прогнозирования дорожного трафика и оценить качество результатов.

В этой главе мы рассмотрим следующие основные темы:

- «Исследование набора данных PeMS-M»,
- «Обработка набора данных»,
- «Реализация темпоральной GNN».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/tree/main/Chapter15.

Исследование набора данных PeMS-M

В этом разделе мы изучим наш набор данных, чтобы найти закономерности и получить сведения, которые будут полезны для решения интересующей нас задачи.

Набор данных, который мы будем использовать, представляет собой сокращенный вариант набора данных `PeMSD7` [1]. Оригинальный набор данных был получен путем сбора данных о скорости движения с 39 000 сенсорных станций в будние дни мая и июня 2012 года с помощью системы измерения эффективности Caltrans (**Performance Measurement System – PeMS**). В сокращенном варианте мы рассмотрим только 228 станций в 7-м округе Калифорнии. Эти станции выдают 30-секундные измерения скорости, которые в данном наборе данных объединены в 5-минутные интервалы. Например, на следующем рисунке показана система PeMS Caltrans (`pems.dot.ca.gov`) с различными скоростями движения.

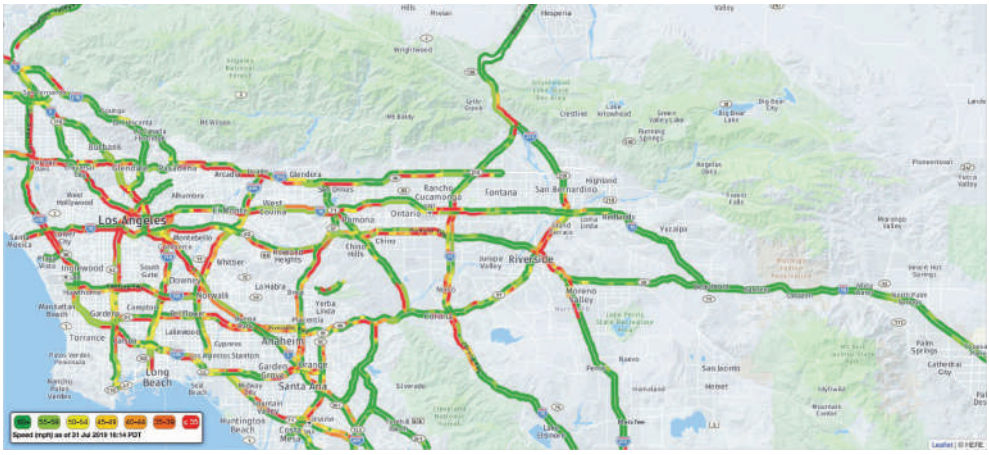


Рис. 15.1 ❖ Данные о дорожном движении Caltrans PeMS, высокая скорость (> 60 миль/ч) обозначена зеленым цветом, низкая скорость (< 35 миль/ч) обозначена красным цветом

Мы можем напрямую загрузить набор данных с GitHub и распаковать его:

```
from io import BytesIO
from urllib.request import urlopen
```

```
from zipfile import ZipFile

url = ("https://github.com/VeritasYin/STGCN_IJCAI-18/"
       "raw/master/dataset/PeMSD7_Full.zip")

with urlopen(url) as zurl:
    with ZipFile(BytesIO(zurl.read())) as zfile:
        zfile.extractall('.')
```

Полученная папка содержит два файла: `V_228.csv` и `W_228.csv`. Файл `V_228.csv` содержит данные о скорости дорожного движения, собранные 228 сенсорными станциями, а файл `W_228.csv` – расстояния между этими станциями.

Давайте загрузим их с помощью pandas. Мы переименуем столбцы с помощью `range()` для удобства доступа:

```
import pandas as pd

speeds = pd.read_csv('PeMSD7_V_228.csv', names=range(0,228))
distances = pd.read_csv('PeMSD7_W_228.csv', names=range(0,228))

speeds
```

	0	1	2	3	4	5	6	7	8	9	...	218	219	220	221	222	223	224	225	226	227
0	71.1	66.0	64.6	65.6	67.1	71.9	68.6	67.7	65.8	40.9	...	69.1	70.9	65.0	64.5	66.6	66.6	65.0	69.3	67.7	68.9
1	68.1	66.8	61.7	66.7	64.5	71.6	72.3	64.9	65.6	40.1	...	70.6	65.4	65.0	64.9	65.1	67.7	65.0	67.7	68.8	68.8
2	68.0	64.3	66.6	68.7	68.1	70.5	70.2	61.7	63.4	39.6	...	72.2	70.5	65.0	64.7	66.7	68.9	65.0	70.2	69.1	68.7
3	68.3	67.8	65.9	66.6	67.9	70.3	69.8	67.6	63.2	37.6	...	71.2	69.7	65.0	65.2	67.2	66.9	65.0	70.4	67.3	69.0
4	68.9	69.5	61.2	67.4	64.0	68.1	67.0	66.7	64.2	36.8	...	71.3	65.8	65.0	66.3	66.7	66.2	65.0	68.0	67.4	68.1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
12667	70.3	65.9	70.2	62.1	66.8	66.0	64.0	64.8	24.6	66.5	...	66.0	55.6	29.9	63.8	64.5	62.8	3.8	70.2	68.1	19.3
12668	69.9	54.0	68.2	62.2	67.4	65.8	64.4	61.3	35.1	69.2	...	66.8	62.8	29.9	62.9	66.6	63.0	3.5	68.2	68.3	18.9
12669	68.9	37.9	68.8	66.3	69.4	66.6	65.0	60.1	38.5	68.7	...	66.4	65.4	29.9	66.3	68.1	62.6	3.5	68.7	67.5	19.7
12670	69.2	37.8	68.7	63.6	68.5	66.2	64.1	60.9	40.5	68.5	...	67.6	68.9	30.2	64.8	68.2	63.1	3.7	67.7	67.4	19.5
12671	68.6	52.9	68.6	65.5	69.3	66.1	64.1	63.5	43.6	58.6	...	66.6	68.3	30.5	64.7	69.1	61.8	4.0	68.1	68.5	19.1

12672 rows × 228 columns

Первое, что мы хотим сделать с этим набором данных, – визуализировать динамику скорости дорожного движения. Это классика прогнозирования временных рядов, поскольку такие характеристики, как сезонность, могут быть чрезвычайно полезны. С другой стороны, нестационарные временные ряды могут потребовать дополнительной обработки, прежде чем их можно будет использовать.

Давайте с помощью matplotlib построим график изменения скорости дорожного движения с течением времени.

1. Мы импортируем NumPy и matplotlib:

```
import numpy as np
import matplotlib.pyplot as plt
```

2. Используем `plt.plot()`, чтобы построить линейные графики:

```
plt.figure(figsize=(10,5), dpi=100)
plt.plot(speeds)
plt.grid(linestyle=':')
plt.xlabel('Время (5 мин)')
plt.ylabel('Скорость дорожного трафика');
```

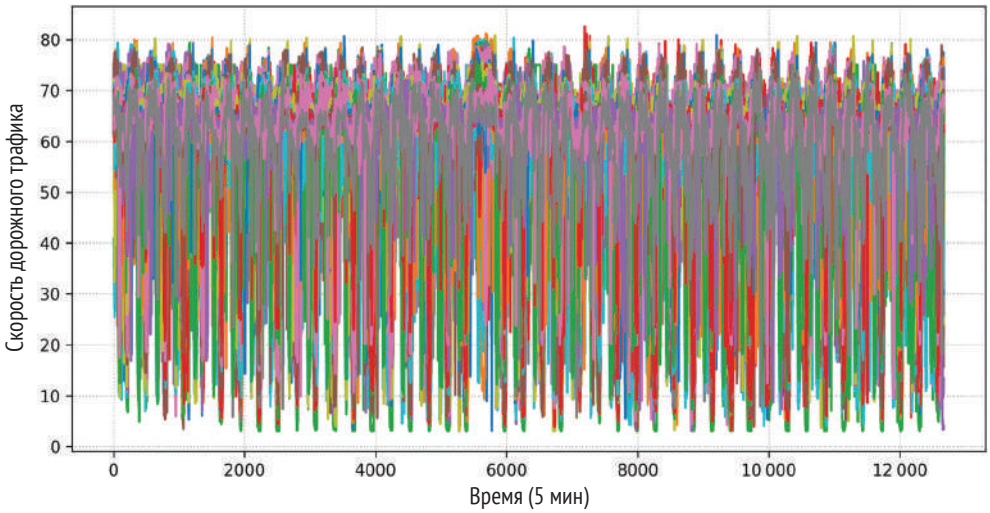


Рис. 15.2 ❖ Изменение скорости дорожного движения с течением времени для каждой из 228 сенсорных станций

К сожалению, данные слишком зашумлены, чтобы мы могли получить какую-либо информацию при таком подходе. Вместо этого мы можем построить графики данных, которые соответствуют нескольким сенсорным станциям. Однако они могут оказаться нерепрезентативными для всего набора данных. Есть и другой вариант: мы можем построить график средней скорости движения со стандартным отклонением. Таким образом, можно представить обобщенную картину набора данных.

На практике мы будем использовать оба подхода, но сейчас попробуем второй вариант.

1. Вычисляем среднюю скорость движения с соответствующим стандартным отклонением для каждого столбца (временного шага):

```
# визуализируем среднюю скорость движения
# со стандартным отклонением
mean = speeds.mean(axis=1)
std = speeds.std(axis=1)
```

2. Мы визуализируем средние значения черной сплошной линией:

```
plt.figure(figsize=(10,5), dpi=100)
plt.plot(mean, 'k-')
```

3. Визуализируем стандартное отклонение вокруг средних значений с помощью функции `plt.fill_between()`, используя светло-красный цвет:

```
plt.grid(linestyle=':')
plt.fill_between(mean.index,
                 mean-std,
                 mean+std,
                 color='r',
                 alpha=0.1)
plt.xlabel('Время (5 мин)')
plt.ylabel('Скорость дорожного трафика');
```

4. Этот программный код генерирует следующий график:

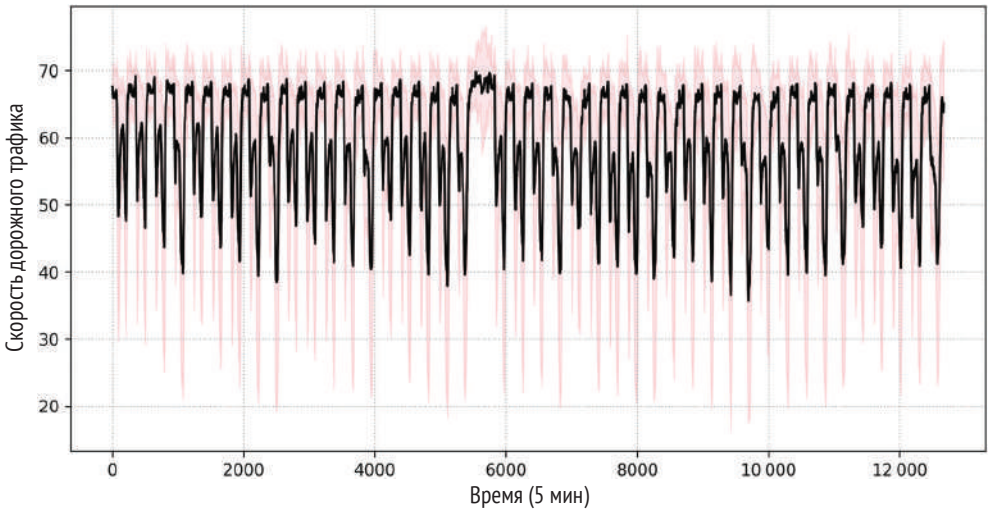


Рис. 15.3 ❖ Усредненная скорость дорожного движения со стандартным отклонением

Этот рисунок гораздо понятнее. Мы видим четкую сезонность (закономерность) в данных временного ряда, за исключением 5800-го наблюдения. Скорость движения имеет большую изменчивость с большими скачками. Это вполне объяснимо, поскольку станции датчиков разбросаны по всему округу 7 Калифорнии: движение может быть затруднено для одних датчиков, но не для других.

Мы можем убедиться в этом, построив график корреляции между значениями скорости, полученными от каждого датчика. Кроме того, мы можем сравнить ее с расстоянием между каждой станцией. Станции, расположенные близко друг к другу, должны показывать одинаковые значения чаще, чем удаленные друг от друга.

Давайте сравним эти два графика на одном рисунке.

1. Создаем два графика с небольшим расстоянием между ними:

```
# мы также можем визуализировать корреляцию между
# временными рядами в различных маршрутах
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 8))
fig.tight_layout(pad=3.0)
```

2. Сначала воспользуемся функцией `matshow()`, чтобы построить график матрицы расстояний:

```
ax1.matshow(distances)
ax1.set_xlabel("Сенсорная станция")
ax1.set_ylabel("Сенсорная станция")
ax1.title.set_text("Матрица расстояний")
```

3. Затем вычисляем коэффициенты корреляции Пирсона для каждой сенсорной станции. Мы должны транспонировать матрицу скоростей, иначе вместо нее получим коэффициенты корреляции для каждого временного шага. Наконец, мы инвертируем их, чтобы оба графика было легче сравнивать:

```
ax2.matshow(-np.corrcoef(speeds.T))
ax2.set_xlabel("Сенсорная станция")
ax2.set_ylabel("Сенсорная станция")
ax2.title.set_text("Корреляционная матрица")
```

4. Получаем следующий график:

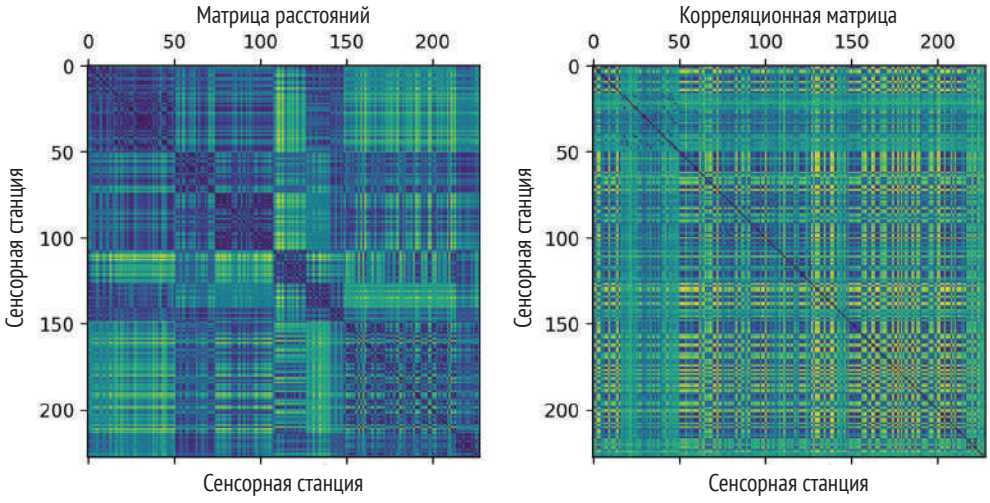


Рис. 15.4 ❖ Матрицы расстояний и корреляций, где более темными цветами обозначены короткие расстояния и высокая корреляция, а более светлыми – большие расстояния и низкая корреляция

Интересно, что большие расстояния между станциями не означают, что они не имеют высокой корреляции (и наоборот). Это особенно важно, если мы рассматриваем только часть этого набора данных: близкие друг к другу станции могут иметь совершенно разные значения, что затрудняет прогнозирование движения. В этой главе мы рассмотрим все сенсорные станции в наборе данных.

Обработка набора данных

Теперь, когда у нас есть больше информации об этом наборе данных, пришло время обработать его, прежде чем подавать в T-GNN.

Первый шаг заключается в преобразовании табличного набора данных в темпоральный граф. Итак, сначала нам нужно создать граф из исходных данных. Другими словами, мы должны связать различные сенсорные станции осмысленным образом. К счастью, у нас есть доступ к матрице расстояний, которая должна стать хорошим способом соединить станции.

Существует несколько вариантов вычисления матрицы смежности на основе матрицы расстояний. Например, мы можем назначить связь, если расстояние между двумя станциями меньше среднего расстояния. Вместо этого мы выполним более сложную обработку, представленную в [2], чтобы вычислить взвешенную матрицу смежности. Вместо бинарных значений мы вычисляем веса от 0 (нет связи) до 1 (сильная связь) по следующей формуле:

$$w_{ij} = \begin{cases} \exp\left(-\frac{d_{ij}^2}{\sigma^2}\right), i \neq j \text{ и } \exp\left(-\frac{d_{ij}^2}{\sigma^2}\right) \geq \epsilon \\ 0 \text{ в противном случае} \end{cases}.$$

Здесь w_{ij} представляет собой вес ребра между узлами i и j, d_{ij} – это расстояние между двумя узлами. Параметры σ^2 и ϵ контролируют распределение и разреженность матрицы смежности. Официальная реализация [2] доступна на GitHub https://github.com/VeritasYin/STGCN_IJCAI-18. Мы воспользуемся теми же самыми значениями $\sigma^2 = 0.1$ и $\epsilon = 0.5$.

Давайте реализуем ее на Python и построим график полученной матрицы смежности.

1. Создаем функцию для вычисления матрицы смежности, которая принимает три параметра: матрицу расстояний и значения σ^2 и ϵ. Как и в официальной реализации, мы делим расстояния на 10 000 и вычисляем d^2:

```
def compute_adj(distances, sigma2=0.1, epsilon=0.5):
    d = distances.to_numpy() / 10000.
    d2 = d * d
```

2. Здесь нам нужны веса, когда их значения больше или равны значению ϵ (в противном случае они должны быть равны нулю). Когда мы проверяем, больше или равно значение веса значению ϵ, результатом будут значения `True` или `False`. Поэтому нам нужна маска из единиц (`w_mask`), чтобы преобразовать их обратно в значения 0 и 1. Мы умножаем веса на маску, чтобы получить только действительные значения весов, которые больше или равны значению ϵ:

```
    n = distances.shape[0]
    w_mask = np.ones([n, n]) - np.identity(n)
    return np.exp(-d2 / sigma2) * (np.exp(-d2 / sigma2) >= epsilon) * w_mask
```

3. Давайте вычислим нашу матрицу смежности и напечатаем результат:

```
adj = compute_adj(distances)
adj[0]
```

```
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.61266012, 0.        , ...
```

Мы видим значение `0.61266012`, представляющее собой вес ребра, идущего от узла 1 к узлу 2.

4. Более эффективным способом визуализации этой матрицы будет повторное использование функции `matshow()` библиотеки matplotlib:

```
plt.figure(figsize=(8, 8))
cax = plt.matshow(adj, False)
plt.colorbar(cax)
plt.xlabel("Сенсорная станция")
plt.ylabel("Сенсорная станция");
```

Мы получаем график, представленный на рис. 15.5.

Визуализированная взвешенная матрица расстояний – это отличный способ подвести итог первого этапа обработки данных. Мы можем сравнить ее с матрицей расстояний, которую мы построили ранее, чтобы найти сходства.

5. Также можно напрямую построить ее в виде графа с помощью функции `draw_networkx()`. В этом случае связи являются бинарными, поэтому мы можем просто считать, что каждый вес больше 0. Мы могли бы отобразить эти значения с помощью меток ребер, но тогда график будет очень трудно читать:

```
import matplotlib.pyplot as plt
import networkx as nx

def plot_graph(adj):
    plt.figure(figsize=(10,5))
    rows, cols = np.where(adj > 0)
    edges = zip(rows.tolist(), cols.tolist())
    G = nx.Graph()
```

```
G.add_edges_from(edges)
nx.draw_networkx(G, with_labels=True)
plt.show()
```

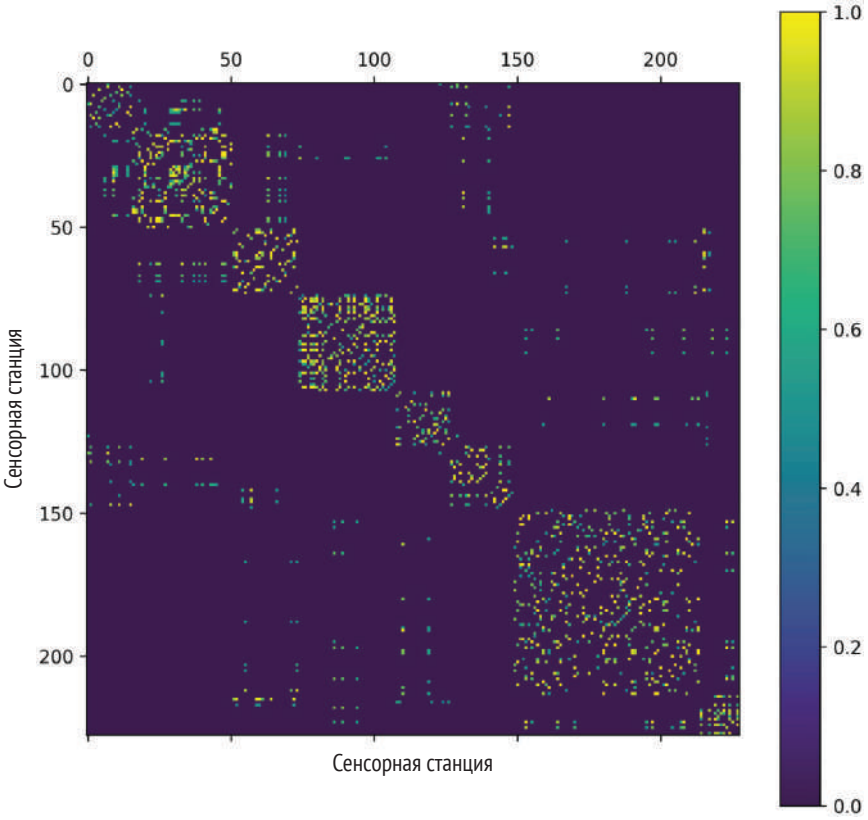


Рис. 15.5 ❖ Взвешенная матрица смежности для набора данных PeMS-M

6. Даже без меток полученный граф трудно проанализировать:

```
plot_graph(adj)
```

В итоге получаем вывод, показанный на рис. 15.6.

Действительно, многие узлы взаимосвязаны, поскольку находятся очень близко друг к другу. Но, несмотря на это, мы можем выделить несколько ветвей, которые могут соответствовать реальным дорогам.

Теперь, когда у нас есть граф, можем сосредоточиться на темпоральном аспекте этой задачи. Первый шаг заключается в нормализации значений скорости, чтобы их можно было передать в нейронную сеть. В литературе по прогнозированию трафика многие авторы выбирают нормализацию по z-значению (или стандартизацию), что мы и сделаем здесь.

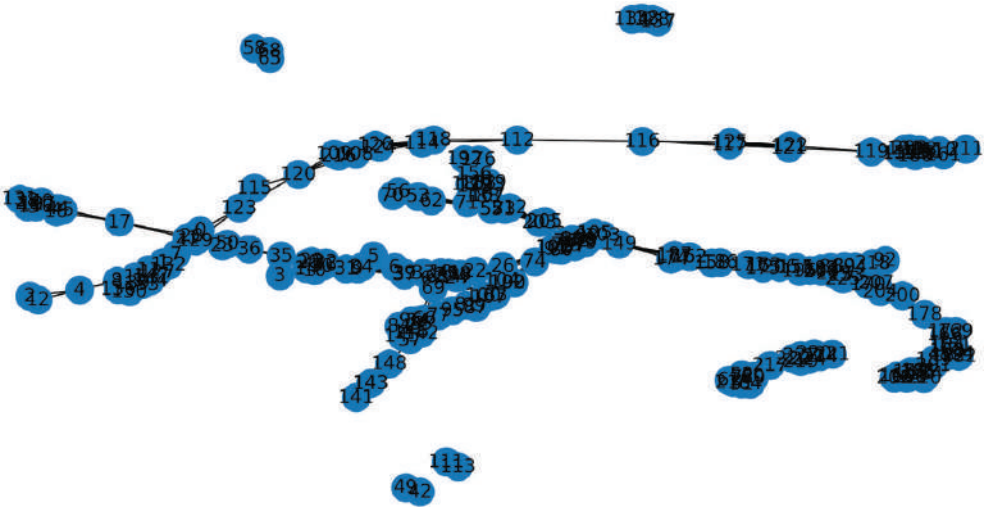


Рис. 15.6 ❖ Набор данных PeMS-M в виде графа (каждый узел представляет собой сенсорную станцию)

1. Мы пишем функцию для вычисления z-значений:

```
# пишем функцию нормализации
def zscore(x, mean, std):
    return (x - mean) / std
```

2. Применяем ее для нормализации данных:

```
# выполняем нормализацию
speeds_norm = zscore(speeds,
                     speeds.mean(axis=0),
                     speeds.std(axis=0))
```

3. Взглянем на результат:

```
speeds_norm.head(1)
```

	0	1	2	3	4	5	6	7	8	9	...	218	219	220	221	222
0	0.950754	0.548255	0.502211	0.831672	0.793696	1.193806	0.983384	0.737268	0.933144	-0.718118	...	0.542628	1.011204	0.609678	0.79198	0.709507

1 rows × 228 columns

Рис. 15.7 ❖ Пример стандартизированных значений скорости

Эти значения корректно стандартизированы. Мы можем использовать их, чтобы создать временной ряд для каждого узла. Нам нужно n входных наблюдений на каждом временном шаге t, чтобы предсказать значение скорости на шаге $t + h$. Большое количество входных данных также увеличивает требования к объему памяти. Значение h, также называемое горизонтом, зависит

от того, какую задачу мы хотим решить – краткосрочное или долгосрочное прогнозирование дорожного движения.

В данном примере возьмем высокое значение 48, чтобы предсказать скорость движения через 4 часа.

1. Мы инициализируем переменные: количество лагов, горизонт, матрицу признаков и матрицу меток:

```
# создаем набор данных
lags = 24
horizon = 48
xs = []
ys = []
```

2. Для каждого временного шага t мы сохраняем 24 предыдущих значения в `xs` и значение на шаге $t + h$ в `ys`:

```
for i in range(lags, speeds_norm.shape[0]-horizon):
    xs.append(speeds_norm.to_numpy()[i-lags:i].T)
    ys.append(speeds_norm.to_numpy()[i+horizon-1])
```

3. Наконец, мы можем создать темпоральный граф с помощью PyTorch Geometric Temporal. Нужно передать индекс ребер в формате COO и вес ребер из взвешенной матрицы смежности:

```
# преобразовываем матрицу смежности в индекс ребер (COO-формат)
edge_index = (np.array(adj) > 0).nonzero()
edge_index

(array([  0,   0,   0, ..., 227, 227, 227]),
 array([  7, 123, 129, ..., 221, 222, 224]))

from torch_geometric_temporal.signal import StaticGraphTemporalSignal

dataset = StaticGraphTemporalSignal(edge_index,
                                    adj[adj > 0],
                                    xs, ys)
```

4. Давайте выведем информацию о первом графе, чтобы проверить, все ли в порядке:

```
dataset[0]

Data(x=[228, 24], edge_index=[2, 1664], edge_attr=[1664], y=[228])
```

5. Не забываем про разбиение на обучающий и тестовый наборы:

```
from torch_geometric_temporal.signal import temporal_signal_split

train_dataset, test_dataset = temporal_signal_split(dataset, train_ratio=0.8)
```

Итоговый темпоральный граф содержит 228 узлов с 12 значениями и 1664 соединениями. Теперь мы готовы применить T-GNN для прогнозирования трафика.

Реализация архитектуры A3T-GCN

В этом разделе мы обучим **темпоральную графовую сверточную сеть с механизмом внимания** (Attention Temporal Graph Convolutional Network – A3T-GCN), предназначенную для прогнозирования трафика. Эта архитектура позволяет учитывать сложные пространственные и темпоральные зависимости.

Пространственные зависимости подразумевают, что на характеристики трафика в одном месте могут влиять характеристики трафика в соседних локациях. Например, пробки часто распространяются на соседние дороги.

Темпоральные зависимости подразумевают, что на характеристики трафика в определенном месте в определенный момент времени могут влиять характеристики трафика в том же месте в предыдущие моменты времени.

A3T-GCN – это улучшенный вариант архитектуры **темпоральной GCN** (**TGCN**). TGCN – это комбинация GCN и GRU, которая генерирует скрытые векторы из каждого входного временного ряда. Комбинация этих двух слоев извлекает пространственную и темпоральную информацию из входного сигнала. Затем используется модель внимания для расчета весов и вывода контекстного вектора. Итоговый прогноз основывается на полученном контекстном векторе. Добавление этой модели внимания обусловлено необходимостью понимания глобальных тенденций.

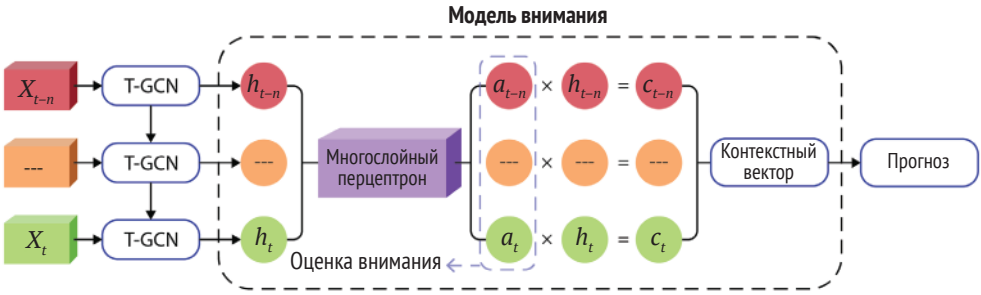


Рис. 15.8 ❖ Архитектура A3T-GCN

Сейчас мы реализуем эту архитектуру с помощью библиотеки PyTorch Geometric Temporal.

1. Сначала импортируем необходимые библиотеки:

```
import torch
from torch_geometric_temporal.nn.recurrent import A3TGCN
```

2. Мы создаем T-GNN с A3TGCN-слоем и линейным слоем с 32 скрытыми измерениями. Параметр `edge_attr` будет хранить веса наших ребер:

```
class TemporalGNN(torch.nn.Module):
    def __init__(self, dim_in, periods):
```

```
        super().__init__()
        self.tgnn = A3TGCN(in_channels=dim_in,
                           out_channels=32,
                           periods=periods)
        self.linear = torch.nn.Linear(32, periods)

    def forward(self, x, edge_index, edge_attr):
        h = self.tgnn(x, edge_index, edge_attr).relu()
        h = self.linear(h)
        return h
```

3. Мы создаем экземпляр класса `TemporalGNN` и задаем оптимизатор Adam с темпом обучения 0.005. Из-за особенностей реализации мы будем обучать эту модель на CPU, а не на GPU, такой вариант обучения в данном случае будет быстрее:

```
model = TemporalGNN(lags, 1).to('cpu')
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
model.train()
print(model)
TemporalGNN(
  (tgnn): A3TGCN(
    (_base_tgcn): TGCN(
      (conv_z): GCNConv(24, 32)
      (linear_z): Linear(in_features=64, out_features=32, bias=True)
      (conv_r): GCNConv(24, 32)
      (linear_r): Linear(in_features=64, out_features=32, bias=True)
      (conv_h): GCNConv(24, 32)
      (linear_h): Linear(in_features=64, out_features=32, bias=True)
    )
  )
  (linear): Linear(in_features=32, out_features=1, bias=True)
)
```

4. Мы обучаем модель, задав 30 эпох, используя среднеквадратичную ошибку (MSE) в качестве функции потерь. Здесь мы используем обратное распространение ошибки после каждой эпохи:

```
# обучение
for epoch in range(30):
    loss = 0
    step = 0
    for i, snapshot in enumerate(train_dataset):
        y_pred = model(snapshot.x.unsqueeze(2),
                       snapshot.edge_index,
                       snapshot.edge_attr)
        loss += torch.mean((y_pred-snapshot.y)**2)
        step += 1
    loss = loss / (step + 1)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

```
    if epoch % 5 == 0:
        print(f"Эпоха {epoch:>2} | MSE на обучении: {loss:.4f}")
```

5. Получаем следующий вывод:

```
Эпоха  0 | MSE на обучении: 0.9945
Эпоха  5 | MSE на обучении: 0.9656
Эпоха 10 | MSE на обучении: 0.9459
Эпоха 15 | MSE на обучении: 0.9279
Эпоха 20 | MSE на обучении: 0.9132
Эпоха 25 | MSE на обучении: 0.9016
```

Теперь, когда наша модель обучена, нам нужно оценить ее качество. Помимо классических показателей, таких как среднеквадратичная ошибка (RMSE) и средняя абсолютная ошибка (MAE), особенно полезно сравнить нашу модель с базовым решением на основе имеющихся временных рядов. В следующем списке мы представим два метода.

Использование **случайного блуждания** (Random Walk – RW) в качестве наивной прогнозной модели. В данном случае под RW подразумевается использование последнего наблюдения в качестве спрогнозированного значения.

Использование **исторического среднего** (Historical Average – HA) в качестве немного более продвинутого решения. В этом случае в качестве значения на временном шаге $t + h$ мы рассчитываем среднюю скорость движения по k предыдущим наблюдениям. В данном примере в качестве значения t мы будем использовать количество лагов, но можно взять и общее среднее историческое значение.

Начнем с оценки качества прогноза модели на тестовом наборе.

1. Пишем функцию для инвертирования z-значений и возврата к исходным значениям (функцию обратного преобразования):

```
def inverse_zscore(x, mean, std):
    return x * std + mean
```

2. Используем ее, чтобы перевести скорости, которые мы хотим предсказать, из нормализованных значений в обычные. Следующий цикл не очень эффективен, но он понятнее, чем более оптимизированный код:

```
y_test = []
for snapshot in test_dataset:
    y_hat = snapshot.y.numpy()
    y_hat = inverse_zscore(y_hat,
                           speeds.mean(axis=0),
                           speeds.std(axis=0))
    y_test = np.append(y_test, y_hat)
```

3. Мы применяем обратное преобразование к прогнозам, полученным с помощью GNN:

```
gnn_pred = []
model.eval()
for snapshot in test_dataset:
    snapshot = snapshot
    y_hat = model(snapshot.x.unsqueeze(2),
                  snapshot.edge_index,
                  snapshot.edge_weight).squeeze().detach().numpy()
    y_hat = inverse_zscore(y_hat,
                           speeds.mean(axis=0),
                           speeds.std(axis=0))
    gnn_pred = np.append(gnn_pred, y_hat)
```

4. Пишем функции для расчета MAE, RMSE и **средней абсолютной процентной ошибки** (Mean Absolute Percentage Error – MAPE):

```
def MAE(real, pred):
    return np.mean(np.abs(pred - real))

def RMSE(real, pred):
    return np.sqrt(np.mean((pred - real) ** 2))

def MAPE(real, pred):
    return np.mean(np.abs(pred - real) / (real + 1e-5))
```

5. Оцениваем качество прогнозов GNN:

```
print(f'GNN MAE  = {MAE(gnn_pred, y_test):.4f}')
print(f'GNN RMSE = {RMSE(gnn_pred, y_test):.4f}')
print(f'GNN MAPE = {MAPE(gnn_pred, y_test):.4f}')
```

```
GNN MAE  = 8.2998
GNN RMSE = 11.9450
GNN MAPE = 0.1490
```

6. Применяем обратное преобразование к прогнозам, полученным с помощью RW и HA, и оцениваем качество прогнозов:

```
rw_pred = []
for snapshot in test_dataset:
    y_hat = snapshot.x[:,-1].squeeze().detach().numpy()
    y_hat = inverse_zscore(y_hat,
                           speeds.mean(axis=0),
                           speeds.std(axis=0))
    rw_pred = np.append(rw_pred, y_hat)

print(f'RW MAE  = {MAE(rw_pred, y_test):.4f}')
print(f'RW RMSE = {RMSE(rw_pred, y_test):.4f}')
print(f'RW MAPE = {MAPE(rw_pred, y_test):.4f}')
```

```
RW MAE  = 11.0469
RW RMSE = 17.6501
RW MAPE = 0.2999
```

```
ha_pred = []
```

```
for i in range(lags, speeds_norm.shape[0]-horizon):
    y_hat = speeds_norm.to_numpy()[:i].T.mean(axis=1)
    y_hat = inverse_zscore(y_hat, speeds.mean(axis=0),
                           speeds.std(axis=0))
    ha_pred.append(y_hat)
ha_pred = np.array(ha_pred).flatten()[-len(y_test):]

print(f'HA MAE  = {MAE(ha_pred, y_test):.4f}')
print(f'HA RMSE = {RMSE(ha_pred, y_test):.4f}')
print(f'HA MAPE = {MAPE(ha_pred, y_test):.4f}')
```

```
HA MAE  = 9.3374
HA RMSE = 13.1518
HA MAPE = 0.1633
```

Мы видим, что модель A3T-GCN превосходит базовые модели по всем метрикам. Это важный результат, поскольку базовые методы часто бывает трудно превзойти. Было бы интересно сравнить эти прогнозы с прогнозами, полученными с помощью LSTM или GRU, чтобы оценить важность топологической информации.

Наконец, мы можем построить график усредненных прогнозов, чтобы получить визуализацию, похожую на рис. 15.3.

1. Мы получаем усредненные прогнозы с помощью генератора списка, который немного быстрее, чем предыдущий метод (но этот код труднее читать):

```
y_preds = [inverse_zscore(
    model(snapshot.x.unsqueeze(2), snapshot.edge_index,
          snapshot.edge_weight).squeeze().detach().numpy(),
    speeds.mean(axis=0),
    speeds.std(axis=0)).mean() for snapshot in test_dataset]
```

2. Вычисляем среднее значение и стандартное отклонение исходного набора данных:

```
mean = speeds.mean(axis=1)
std = speeds.std(axis=1)
```

3. Визуализируем среднюю скорость движения со стандартным отклонением и сравним ее со спрогнозированными значениями ($t + h$):

```
plt.figure(figsize=(10,5))
plt.plot(np.array(mean), 'k-', label='Среднее')
plt.plot(range(len(speeds)-len(y_preds),
               len(speeds)), y_preds,
         'r-',
         label='Прогноз')
plt.grid(linestyle=':')
plt.fill_between(mean.index,
                 mean-std,
                 mean+std,
```

```
                color='r',
                alpha=0.1)
plt.axvline(x=len(speeds)-len(y_preds),
            color='b',
            linestyle='--')
plt.xlabel('Время (5 мин)')
plt.ylabel('Прогнозируемая скорость движения')
plt.legend(loc='upper right');
```

Получаем следующий график:

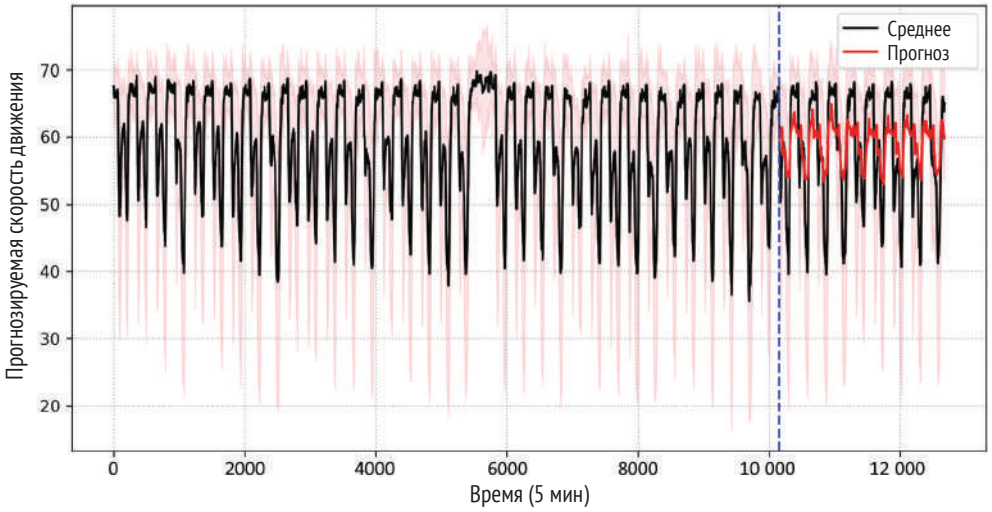


Рис. 15.9 ❖ Средняя скорость дорожного движения, предсказанная моделью A3T-GCN на тестовом наборе

T-GNN правильно предсказывает пики и следует общему тренду. Однако предсказанные скорости ближе к общему среднему значению, так как из-за функции потерь MSE модели более затратно совершать серьезные ошибки. Несмотря на это, модель GNN достаточна точна и может быть настроена для получения более экстремальных значений.

Выводы

В этой главе мы рассмотрели задачу прогнозирования дорожного трафика с помощью T-GNN. Сначала мы исследовали набор данных PeMS-M и преобразовали его из табличных данных в статический графовый набор данных с темпоральным сигналом. Мы создали взвешенную матрицу смежности на основе входной матрицы расстояний и преобразовали скорости движения во временные ряды. Наконец, мы реализовали модель A3T-GCN, T-GNN, предна-

значенную для прогнозирования трафика. Мы сравнили полученные результаты с результатами двух базовых моделей и проверили качество прогнозов, сделанных нашей моделью.

В главе 16 «Построение рекомендательной системы с помощью LightGCN» мы рассмотрим наиболее популярный пример применения GNN. Мы реализуем облегченную модель GNN на большом наборе данных и оценим ее с помощью методов, используемых в рекомендательных системах.

Дополнительное чтение

[1] B. Yu, H. Yin, and Z. Zhu. *Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting*. Jul. 2018. doi: 10.24963/ijcai.2018/505. Доступ по ссылке https://arxiv.org/abs/1709.04875.

[2] Y. Li, R. Yu, C. Shahabi, and Y. Liu. *Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting*. arXiv, 2017. doi: 10.48550/ARXIV.1707.01926. Доступ по ссылке https://arxiv.org/abs/1707.01926.

Глава 16

Построение рекомендательной системы с помощью LightGCN

Рекомендательные системы стали неотъемлемой частью современных онлайн-платформ, цель которых – предоставлять пользователям персонализированные рекомендации на основе их интересов и прошлых взаимодействий. Эти системы можно встретить в самых разных приложениях, включая рекомендации товаров для покупки на сайтах электронной коммерции, рекомендации контента для просмотра на потоковых сервисах и предложения завести знакомства в социальных сетях.

Рекомендательные системы – один из основных примеров применения GNN. Действительно, они могут эффективно включать в единую модель сложные отношения между пользователями, товарами и их взаимодействиями. Кроме того, структура графа позволяет включать в процесс генерации рекомендаций побочную информацию, такую как метаданные пользователей и товаров.

В этой главе мы представим новую архитектуру GNN под названием LightGCN, специально разработанную для рекомендательных систем. Мы также представим новый набор данных Book-Crossing, который содержит пользователей, книги и более миллиона рейтингов. Используя этот набор данных, мы построим систему рекомендаций книг с использованием коллаборативной фильтрации и применим ее с целью получения рекомендаций для конкретного пользователя. В ходе этого процесса мы продемонстрируем, как использовать архитектуру LightGCN для создания практической рекомендательной системы.

К концу этой главы вы сможете создать свою собственную рекомендательную систему с помощью LightGCN. Вы узнаете, как обрабатывать любые

наборы данных с пользователями, товарами и рейтингами для последующего применения коллаборативной фильтрации. Наконец, вы узнаете, как реализовать и оценить качество этой архитектуры и получить рекомендации для отдельных пользователей.

В этой главе мы рассмотрим следующие основные темы:

- «Исследование набора данных Book-Crossing»,
- «Предварительная обработка набора данных Book-Crossing»,
- «Реализация архитектуры LightGCN».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/blob/main/Chapter16/Chapter16.ipynb.

Исследование набора данных Book-Crossing

В этом разделе мы проведем разведочный анализ нового набора данных и визуализируем его основные характеристики.

Набор данных `Book-Crossing` [1] представляет собой коллекцию рейтингов книг, выставленных 278 858 пользователями сообщества `BookCrossing` (www.bookcrossing.com). В общей сложности 1 149 780 рейтингов, как явных (оценка от 1 до 10), так и неявных (пользователи взаимодействовали с книгой), относятся к 271 379 книгам. Набор данных был собран Цай-Николасом Зиглером в течение четырех недель в августе и сентябре 2004 года. В этой главе мы будем использовать набор данных `Book-Crossing` для создания системы рекомендаций книг.

Мы будем работать с тремя файлами.

- Файл `BX-Users.csv` содержит данные об отдельных пользователях Book-Crossing. Идентификаторы пользователей были анонимизированы и представлены в виде целых чисел. Для некоторых пользователей также включена демографическая информация, такая как местоположение и возраст. Если эта информация недоступна, соответствующие поля содержат значения `NULL`.
- Файл `BX-Books.csv` содержит данные о книгах, включенных в набор данных и идентифицированных по их ISBN. Недействительные ISBN были удалены из набора данных. Помимо информации о содержании типа названия книги, автора, года издания и издателя этот файл также содержит URL-адреса, ссылающиеся на изображения обложек книг трех разных размеров.

- Файл `BX-Book-Ratings.csv` содержит информацию о рейтингах, присвоенных книгам в наборе данных. Оценки могут быть как явными, выставляемыми по шкале от 1 до 10, где более высокие значения указывают на более высокую оценку, так и неявными, когда рейтинг равен 0.

На рис. 16.1 представлен граф, построенный с помощью Gephi на основе подвыборки из этого набора данных.

Рис. 16.1 ❖ Представление набора данных Book-Crossing в виде графа, в котором книги представлены в виде синих узлов, а пользователи – в виде красных узлов

Размер узлов пропорционален количеству связей (степени) в графе. Мы можем видеть популярные книги, такие как «Код да Винчи», которые выполняют роль хабов благодаря большому количеству связей.

Теперь давайте исследуем набор данных, чтобы получить больше информации.

1. Импортируем pandas и загружаем каждый файл с разделителем ; и кодировкой `latin-1` для совместимости. `BX-Books.csv` также требует параметра `error_bad_lines`:

```
import pandas as pd

ratings = pd.read_csv("BX-Book-Ratings.csv",
                      sep=';', encoding='latin-1')
users = pd.read_csv("BX-Users.csv",
                    sep=';', encoding='latin-1')
books = pd.read_csv("BX-Books.csv", sep=';',
                    encoding='latin-1', error_bad_lines=False)
```

2. Давайте выведем эти наборы данных, чтобы увидеть количество строк и столбцов:

```
ratings
```

	User-ID	ISBN	Book-Rating
0	276725	034545104X	0
1	276726	0155061224	5
2	276727	0446520802	0
3	276729	052165615X	3
4	276729	0521795028	6
...	...	...	...
1149775	276704	1563526298	9
1149776	276706	0679447156	0
1149777	276709	0515107662	10
1149778	276721	0590442449	10
1149779	276723	05162443314	8

1149780 rows × 3 columns

```
users
```

	User-ID	Location	Age
0	1	nyc, new york, usa	NaN
1	2	stockton, california, usa	18.0
2	3	moscow, yukon territory, russia	NaN
3	4	porto, v.n.gaia, portugal	17.0
4	5	farnborough, hants, united kingdom	NaN
...	...	...	...
278853	278854	portland, oregon, usa	NaN
278854	278855	tacoma, washington, united kingdom	50.0
278855	278856	brampton, ontario, canada	NaN
278856	278857	knoxville, tennessee, usa	NaN
278857	278858	dublin, n/a, ireland	NaN

278858 rows × 3 columns

books

	ISBN	Book-Title	Book-Author	Year-Of-Publication	Publisher	Image-URL-S
0	0195153448	Classical Mythology	Mark P. O. Morford	2002	Oxford University Press	http://images.amazon.com/images/P/0195153448.0...
1	0002005018	Clara Callan	Richard Bruce Wright	2001	HarperFlamingo Canada	http://images.amazon.com/images/P/0002005018.0...
2	0060973129	Decision in Normandy	Carlo D'Este	1991	HarperPerennial	http://images.amazon.com/images/P/0060973129.0...
3	0374157065	Flu: The Story of the Great Influenza Pandemic...	Gina Bari Kolata	1999	Farrar Straus Giroux	http://images.amazon.com/images/P/0374157065.0...
4	0393045218	The Mummies of Urumchi	E. J. W. Barber	1999	W. W. Norton & Company	http://images.amazon.com/images/P/0393045218.0...
...	...	...	...	...	...	...
271374	0440400988	There's a Bat in Bunk Five	Paula Danziger	1988	Random House Childrens Pub (Mm)	http://images.amazon.com/images/P/0440400988.0...
271375	0525447644	From One to One Hundred	Teri Sloat	1991	Dutton Books	http://images.amazon.com/images/P/0525447644.0...
271376	006008667X	Lily Dale : The True Story of the Town that Ta...	Christine Wicker	2004	HarperSanFrancisco	http://images.amazon.com/images/P/006008667X.0...
271377	0192126040	Republic (World's Classics)	Plato	1996	Oxford University Press	http://images.amazon.com/images/P/0192126040.0...
271378	0767409752	A Guided Tour of Rene Descartes' Meditations o...	Christopher Biffle	2000	McGraw-Hill Humanities/Social Sciences/Languages	http://images.amazon.com/images/P/0767409752.0...

271379 rows × 8 columns

3. Датафрейм `ratings` соединяет датафреймы `users` и `books`, используя информацию об идентификаторе пользователя (`User-ID`) и ISBN (`ISBN`), и включает рейтинг, который можно считать весом. Датафрейм `users` включает демографическую информацию, такую как местоположение и возраст, для каждого пользователя, если она доступна. Датафрейм `books` включает информацию о содержании книг: название, автора, год издания, издателя, а также URL-адреса, ведущие на изображения обложек трех разных размеров.
4. Давайте визуализируем распределение рейтингов, чтобы понять, можем ли мы использовать эту информацию. Мы можем построить график с помощью matplotlib и seaborn следующим образом:

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.countplot(x=ratings['Book-Rating']);
```

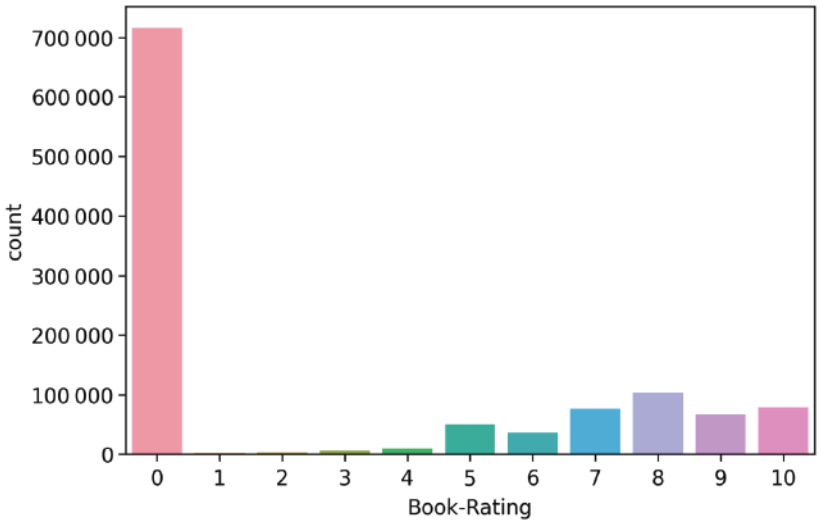


Рис. 16.2 ❖ Распределение рейтингов
(взаимодействие с книгой представлено в виде нулевого рейтинга,
а рейтинги от 1 до 10 – это реальные рейтинги)

5. Соответствуют ли эти рейтинги данным в датафреймах `books` и `users`? В качестве быстрой проверки мы можем сравнить количество уникальных записей `User-ID` и `ISBN` в датафрейме `ratings` с количеством строк в этих датафреймах:

```
print(len(ratings['User-ID'].unique()))
print(len(ratings['ISBN'].unique()))
```

```
105283
340556
```

   Интересно, что в датафрейме `ratings` меньше уникальных пользователей по сравнению с датафреймом `users` (105 283 против 278 858), но больше уникальных ISBN по сравнению датафремом `books` (340 556 против 271 379). Это означает, что в нашей базе данных не хватает многих значений, поэтому нужно быть осторожными при объединении таблиц.

6. В завершение построим график количества книг, которые были оценены только один раз, два раза и т. д. Сначала подсчитаем количество появлений каждого уникального ISBN в датафрейме `ratings`, используя методы `.groupby()` и `.size()`:

```
isbn_counts = ratings.groupby('ISBN').size()
```

   В результате создаем новый датафрейм `isbn_counts`, который содержит частоту каждого уникального ISBN в датафрейме `ratings`.

7. С помощью метода `.value_counts()` подсчитаем количество появлений каждого уникального значения в датафрейме `isbn_counts`, который мы

только что создали. Результат сохраняем в новом датафрейме `count_occurrences`. Таким образом, получаем информацию о том, сколько книг имеют только одну оценку, сколько книг имеют две оценки и т. д., в зависимости от количества оценок, которые они получили:

```
count_occurrences = isbn_counts.value_counts()
```

8. Наконец, мы можем построить график распределения с помощью метода pandas `.plot()`. В данном случае построим график только первых 15 значений:

```
count_occurrences[:15].plot(kind='bar')
plt.xlabel("Количество вхождений номера ISBN")
plt.ylabel("Частота");
```

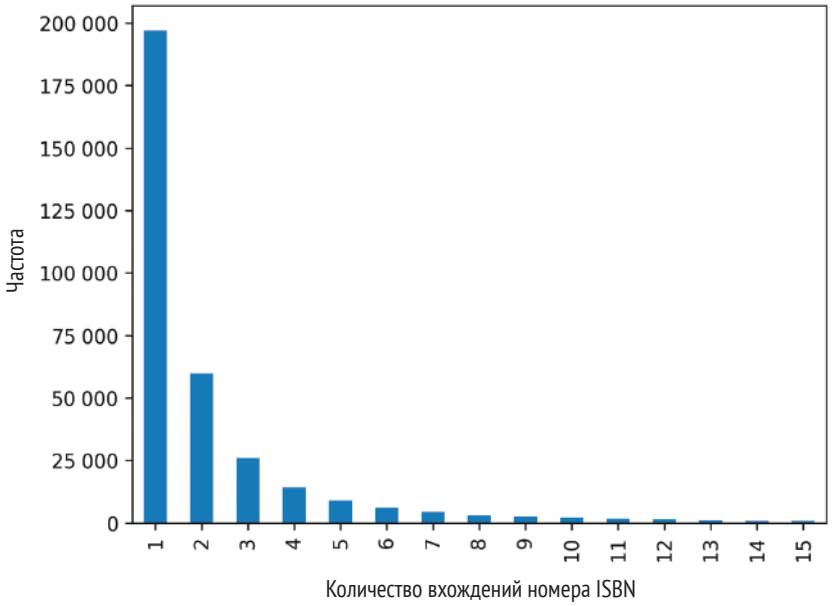


Рис. 16.3 ❖ Распределение количества появлений каждой книги (ISBN) в рейтингах (первые 15 значений)

Мы видим, что многие книги были оценены всего один или два раза. Очень редко встречаются книги с большим количеством оценок, что усложняет нашу работу, поскольку мы полагаемся на эти зависимости.

9. Повторяем аналогичные операции, чтобы получить распределение количества раз, когда каждый пользователь (`User-ID`) появляется в датафрейме `ratings`:

```
userid_counts = ratings.groupby('User-ID').size()
count_occurrences = userid_counts.value_counts()
count_occurrences[:15].plot(kind='bar')
```

```
plt.xlabel("Количество вхождений User-ID")
plt.ylabel("Частота");
```

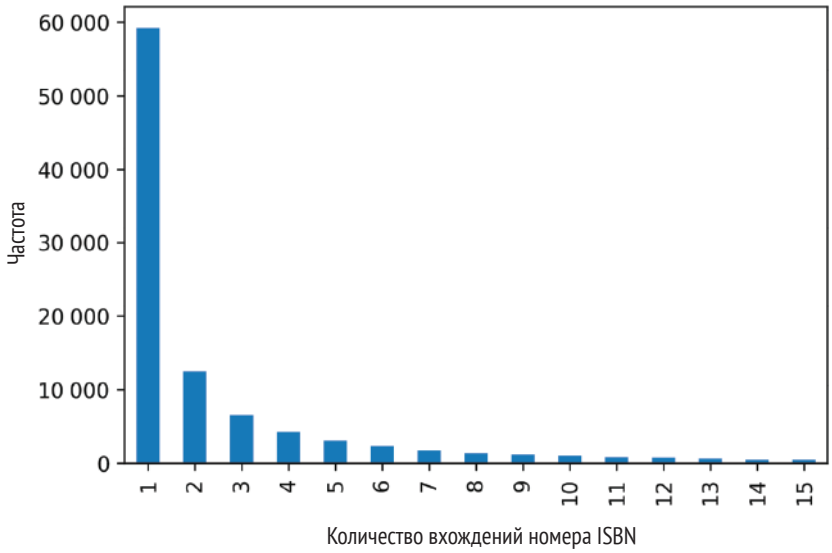


Рис. 16.4 ❖ Распределение количества появлений каждого пользователя (User-ID) в рейтингах (первые 15 значений)

Это также означает, что большинство пользователей оценивают только одну или две книги, но некоторые из них оценивают много книг.

С этим набором данных связаны различные проблемы, например ошибки в годе издания или названии издательства, а также другие недостающие или неверные значения. Однако в этой главе мы не будем напрямую использовать метаданные из датафреймов `books` и `users`. Мы будем полагаться на связи между значениями `User-ID` и `ISBN`, поэтому здесь нам не нужно чистить набор данных.

В следующем разделе рассмотрим, как обработать набор данных, чтобы подготовить его перед отправкой в LightGCN.

Предварительная обработка набора данных Book-Crossing

Мы хотим обработать этот набор данных для решения конкретной задачи: рекомендовать товары, а точнее, использовать подход **коллаборативной фильтрации** (collaborative filtering). Коллаборативная фильтрация – это метод, используемый для составления персонализированных рекомендаций для пользователей. В его основе лежит идея о том, что пользователи, имею-

щие схожие предпочтения или поведение, с большей вероятностью будут иметь схожие интересы. Алгоритмы **коллаборативной фильтрации** (collaborative filtering) используют эту информацию для выявления закономерностей и дают рекомендации пользователям, основываясь на предпочтениях похожих пользователей.

Этот метод отличается от метода фильтрации на основе контента, который представляет собой подход к составлению рекомендаций, основанный на характеристиках рекомендуемых товаров. Он генерирует рекомендации, определяя характеристики товара и сопоставляя их с характеристиками других товаров, которые нравились пользователю в прошлом. Методы фильтрации на основе контента обычно основаны на идее, что если пользователю понравился товар с определенными характеристиками, то ему также понравятся товары с аналогичными характеристиками.

В этой главе мы сосредоточимся на коллаборативной фильтрации. Наша цель – определить, какую книгу рекомендовать пользователю, основываясь на предпочтениях других пользователей. Эту задачу можно представить в виде двудольного графа (биграфа), как показано на рис. 16.5.

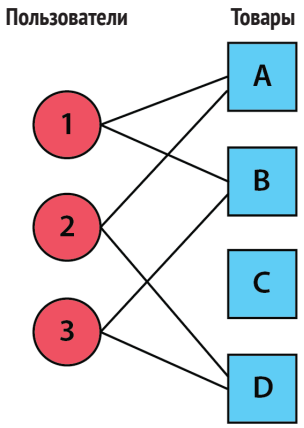


Рис. 16.5 ❖ Пример двудольного графа пользователь–товар

Зная, что пользователю **1** понравились товары **A** и **B**, а пользователю **3** – товары **B** и **D**, мы, вероятно, должны рекомендовать товар **B** пользователю **2**, которому также понравились товары **A** и **D**.

Именно такой тип графа мы хотим построить на основе набора данных `Book-Crossing`. Более точно – мы также хотим включить отрицательные примеры. Здесь под отрицательными примерами понимаются товары, которые не были оценены данным пользователем. Товары, которые были оценены конкретным пользователем, называются положительными. Мы объясним, почему используем эту технику отрицательного семплирования, когда будем реализовывать функцию потерь.

В остальной части главы код LightGCN преимущественно опирается на официальную реализацию, а также на превосходную работу Хотты и Чжоу (Hotta and Zhou [2]) и работу Ли и коллег (Li et al. [3]).

1. Импортируем следующие библиотеки:

```
import numpy as np
from sklearn.model_selection import train_test_split

import torch.nn.functional as F
from torch import nn, optim, Tensor

from torch_geometric.utils import structured_negative_sampling
from torch_geometric.nn.conv.gcn_conv import gcn_norm
from torch_geometric.nn import LGConv
```

2. Заново загружаем наборы данных:

```
df = pd.read_csv('BX-Book-Ratings.csv', sep=';', encoding='latin-1')
users = pd.read_csv('BX-Users.csv', sep=';', encoding='latin-1')
books = pd.read_csv('BX-Books.csv', sep=';', encoding='latin-1',
                    error_bad_lines=False)
```

3. Сохраняем только те строки, в которых значение столбца `ISBN` присутствует в списке уникальных значений столбца `ISBN` датафрейма `books` и значение столбца `User-ID` присутствует в списке уникальных значений столбца `User-ID` датафрейма `users`:

```
# предварительная обработка
df = df.loc[df['ISBN'].isin(books['ISBN'].unique()) &
            df['User-ID'].isin(users['User-ID'].unique())]
```

4. Сохраняем только высокие рейтинги (>= 8/10), чтобы созданные нами связи соответствовали книгам, которые понравились пользователям. Затем отфильтровываем еще больше наблюдений и сохраняем ограниченное количество строк (100 000), чтобы ускорить обучение:

```
# сохраняем 100k наивысших рейтингов
df = df[df['Book-Rating'] >= 8].iloc[:100000]
```

5. Создаем отображения идентификаторов пользователей и товаров в целочисленные индексы:

```
# создаем отображения пользователей и товаров в целочисленные индексы
user_mapping = {userid: i for i, userid in enumerate(df['User-ID'].unique())}
item_mapping = {isbn: i for i, isbn in enumerate(df['ISBN'].unique())}
```

6. Подсчитываем количество пользователей, товаров и общее количество сущностей в наборе данных:

```
# подсчитываем количество пользователей и товаров
num_users = len(user_mapping)
```

```
num_items = len(item_mapping)
num_total = num_users + num_items
```

7. Создаем тензоры из индексов пользователей и товаров на основе оценок пользователей в наборе данных. Тензор `edge_index` создается путем состыковки этих двух тензоров:

```
# создаем матрицу смежности на основе рейтингов пользователей
user_ids = torch.LongTensor([user_mapping[i] for i in df['User-ID']])
item_ids = torch.LongTensor([item_mapping[i] for i in df['ISBN']])
edge_index = torch.stack((user_ids, item_ids))
```

8. Разбиваем `edge_index` на обучающую, проверочную и тестовую части с помощью функции `train_test_split()` библиотеки scikit-learn:

```
# создаем обучающую, проверочную и тестовую матрицы смежности
train_index, test_index = train_test_split(
    range(len(df)), test_size=0.2, random_state=0)
val_index, test_index = train_test_split(
    test_index, test_size=0.5, random_state=0)

train_edge_index = edge_index[:, train_index]
val_edge_index = edge_index[:, val_index]
test_edge_index = edge_index[:, test_index]
```

9. Генерируем батч случайных индексов с помощью функции `np.random.choice()`. Она генерирует `BATCH_SIZE` случайных индексов из диапазона от `0` до `edge_index.shape[1]-1`. Эти индексы будут использоваться для отбора строк из тензора `edge_index`:

```
def sample_mini_batch(edge_index):
    # генерируем BATCH_SIZE случайных индексов
    index = np.random.choice(range(edge_index.shape[1]), size=BATCH_SIZE)
```

10. Генерируем отрицательные примеры с помощью функции `structured_negative_sampling()` из PyTorch Geometric. Отрицательные примеры – это товары, с которыми соответствующий пользователь не взаимодействовал. Мы воспользуемся функцией `torch.stack()`, чтобы объединить по оси 0:

```
    # генерируем индексы отрицательных примеров
    edge_index = structured_negative_sampling(edge_index)
    edge_index = torch.stack(edge_index, dim=0)
```

11. Отбираем индексы пользователя, положительного товара и отрицательного товара для батча, используя массив `index` и тензор `edge_index`:

```
    user_index = edge_index[0, index]
    pos_item_index = edge_index[1, index]
    neg_item_index = edge_index[2, index]

    return user_index, pos_item_index, neg_item_index
```

Тензор `user_index` содержит индексы пользователей для батча, тензор `pos_item_index` – индексы положительных товаров для батча, а тензор `neg_item_index` – индексы отрицательных товаров для батча.

Теперь у нас есть три набора и функция, возвращающая мини-батчи. Следующий шаг – понять и реализовать архитектуру LightGCN.

Реализация архитектуры LightGCN

Архитектура LightGCN [4] нацелена на обучение представлений для узлов путем сглаживания значений признаков по графу. Она итеративно выполняет свертку графа, в ходе которой характеристики соседних узлов объединяются как новое представление целевого узла. Общая схема архитектуры приведена на рис. 16.6.

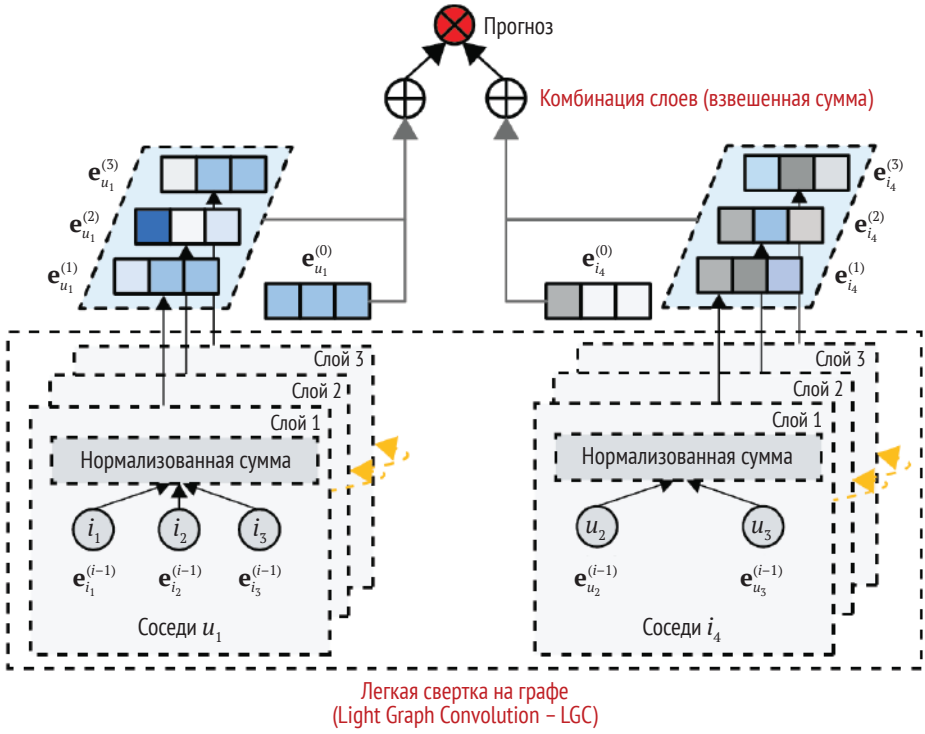


Рис. 16.6 ❖ Архитектура модели LightGCN со сверткой и комбинацией слоев

Однако в LightGCN используется простой агрегатор на основе взвешенной суммы, а не преобразование признаков или нелинейная активация, как в других моделях типа GCN или GAT. Операция легкой свертки на гра-

фе вычисляет эмбеддинги $(k + 1)$-го пользователя и товара $\mathbf{e}_u^{(k+1)}$ и $\mathbf{e}_i^{(k+1)}$ следующим образом:

$$\mathbf{e}_u^{(k+1)} = \sum_{i \in \mathcal{N}_u} \frac{1}{\sqrt{|\mathcal{N}_u|}\sqrt{|\mathcal{N}_i|}} \mathbf{e}_i^{(k)};$$

$$\mathbf{e}_i^{(k+1)} = \sum_{u \in \mathcal{N}_i} \frac{1}{\sqrt{|\mathcal{N}_i|}\sqrt{|\mathcal{N}_u|}} \mathbf{e}_u^{(k)}.$$

Симметричная нормализации гарантирует, что масштаб эмбеддингов не увеличивается при операциях графовой свертки. В отличие от других моделей LightGCN объединяет только связанных соседей и не включает самопетли.

Фактически тот же эффект достигается за счет использования операции комбинации слоев. Этот механизм состоит из взвешенной суммы эмбеддингов пользователя и товара в каждом слое. В результате получаются итоговые эмбеддинги $\mathbf{e}_u$ и $\mathbf{e}_i$, которые определяются следующими уравнениями:

$$\mathbf{e}_u = \sum_{k=0}^{K} \alpha_k \mathbf{e}_u^{(k)};$$

$$\mathbf{e}_i = \sum_{k=0}^{K} \alpha_k \mathbf{e}_i^{(k)}.$$

Здесь вклад k-го слоя взвешивается по переменной $\alpha_k \geq 0$. Авторы LightGCN рекомендуют установить ее равной $1/(K + 1)$, где K – количество слоев.

Прогноз, показанный на рис. 16.6, соответствует рейтингам или ранговым оценкам. Он получается с помощью внутреннего произведения итоговых представлений пользователя и товара:

$$\hat{y}_{ui} = \mathbf{e}_u^T \mathbf{e}_i.$$

Теперь давайте реализуем эту архитектуру в PyTorch Geometric.

1. Создаем класс `LightGCN` с четырьмя аргументами: `num_users`, `num_items`, `num_layers` и `dim_h`. Аргументы `num_users` и `num_items` задают количество пользователей и товаров в наборе данных соответственно. Аргумент `num_layers` указывает количество слоев LightGCN, которые будут использоваться, а аргумент `dim_h` задает размерность эмбеддингов (для пользователей и товаров):

```
class LightGCN(nn.Module):
    def __init__(self, num_users, num_items, num_layers=4, dim_h=64):
        super().__init__()
```

2. Сохраняем количество пользователей и товаров и создаем слои эмбеддингов пользователей и товаров. Форма `emb_users` или $\mathbf{e}_u^{(0)}$ – `(num_users, dim_h)`, форма `emb_items` или $\mathbf{e}_i^{(0)}$ – `(num_items, dim_h)`:

```
        self.num_users = num_users
        self.num_items = num_items
        self.num_layers = num_layers
        self.emb_users = nn.Embedding(
            num_embeddings=self.num_users, embedding_dim=dim_h
        )
        self.emb_items = nn.Embedding(
            num_embeddings=self.num_items, embedding_dim=dim_h
        )
```

3. Создаем список `num_layers` (обозначенных ранее как K) LightGCN-слоев с помощью класса `LGConv` библиотеки PyTorch Geometric. Он будет использоваться для выполнения операций легкой свертки на графе:

```
        self.convs = nn.ModuleList(LGConv() for _ in range(num_layers))
```

4. Инициализируем слои эмбеддингов пользователей и товаров нормальными распределениями со стандартным отклонением 0.01. Это помогает предотвратить застревание модели в плохом локальном оптимуме во время обучения:

```
        nn.init.normal_(self.emb_users.weight, std=0.01)
        nn.init.normal_(self.emb_items.weight, std=0.01)
```

5. Метод `.forward()` принимает тензор `edge_index` и возвращает итоговые эмбеддинги пользователя и товара $\mathbf{e}_u^{(K)}$ и $\mathbf{e}_i^{(K)}$. Он начинает с конкатенации слоев эмбеддингов пользователя и товара и сохраняет результат в тензоре `emb`. Затем создается список `embs`, первым элементом которого является `emb`:

```
    def forward(self, edge_index):
        emb = torch.cat(
            [self.emb_users.weight, self.emb_items.weight]
        )
        embs = [emb]
```

6. Запускаем цикл сверток по LightGCN-слоям и сохраняем выход каждого слоя в списке `embs`:

```
        for conv in self.convs:
            emb = conv(x=emb, edge_index=edge_index)
            embs.append(emb)
```

7. Выполняем комбинирование слоев, вычисляя итоговые эмбеддинги. Для этого вычисляем среднее значение тензоров в списке `embs` по второй размерности и умножаем на $1/(K + 1)$, где K – количество слоев:

```
        emb_final = 1/(self.num_layers+1) * torch.mean(
            torch.stack(embs, dim=1), dim=1
        )
```

8. Разбиваем `emb_final` на эмбеддинги пользователя и товара ($\mathbf{e}_u$ и $\mathbf{e}_i$) и возвращаем их вместе с $\mathbf{e}_u^{(0)}$ и $\mathbf{e}_i^{(0)}$:

```
emb_users_final, emb_items_final = torch.split(
    emb_final, [self.num_users, self.num_items]
)

return (
    emb_users_final,
    self.emb_users.weight,
    emb_items_final,
    self.emb_items.weight
)
```

Для обучения модели нам потребуется функция потерь. В архитектуре LightGCN используется функция потерь **байесовское персонализированное ранжирование (**Bayesian Personalized Ranking – BPR), которая оптимизирует способность модели ранжировать положительные объекты выше, чем отрицательные, для данного пользователя. Ее можно реализовать следующим образом:

$$L_{\text{BPR}} = -\sum_{u=1}^{M} \sum_{i \in \mathcal{N}_u} \sum_{j \notin \mathcal{N}_u} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \lambda \left\| \mathbf{E}^{(0)} \right\|^2.$$

Здесь $\mathbf{E}^{(0)}$ – матрица эмбеддингов 0-го слоя (результат конкатенации исходных эмбеддингов пользователя и товара), λ – сила регуляризации, $\hat{y}_{ui}$ – предсказанный рейтинг положительного товара, $\hat{y}_{uj}$ – предсказанный рейтинг отрицательного товара.

Мы реализуем эту функцию потерь в PyTorch с помощью функции `bpr_loss()`.

1. Вычисляем функцию потерь с регуляризацией на основе эмбеддингов, которые хранятся в модели LightGCN:

```
def bpr_loss(emb_users_final, emb_users, emb_pos_items_final,
             emb_pos_items, emb_neg_items_final, emb_neg_items):
    reg_loss = LAMBDA * (
        emb_users.norm().pow(2) +
        emb_pos_items.norm().pow(2) +
        emb_neg_items.norm().pow(2)
    )
```

2. Вычисляем рейтинги для положительных и отрицательных товаров как скалярное произведение эмбеддингов пользователей и эмбеддингов товаров:

```
    pos_ratings = torch.mul(
        emb_users_final, emb_pos_items_final).sum(dim=-1)
    neg_ratings = torch.mul(
        emb_users_final, emb_neg_items_final).sum(dim=-1)
```

3. В отличие от логарифмического сигмоида в предыдущем уравнении мы вычисляем функцию потерь BPR как среднее значение функции `softplus`, примененной к разнице между положительными и отрицательными рейтингами. Этот вариант был выбран по причине того, что он дал лучшие экспериментальные результаты:

```
bpr_loss = torch.mean(
    torch.nn.functional.softplus(pos_ratings - neg_ratings)
)
# bpr_loss = torch.mean(
#     torch.nn.functional.logsigmoid(pos_ratings - neg_ratings)
#)
```

4. Мы возвращаем функцию потерь BPR с регуляризацией следующим образом:

```
return -bpr_loss + reg_loss
```

Помимо функции потерь BPR, мы воспользуемся двумя метриками для оценки качества нашей модели.

Recall@k – доля релевантных рекомендованных товаров в топ-*k* рекомендациях среди общего количества релевантных товаров. Однако эта метрика не учитывает порядок релевантных товаров в топ-*k* рекомендациях:

$$\text{Recall@k} = \frac{\text{Количество релевантных товаров, найденных в топ-}k\text{ рекомендациях}}{\text{Общее количество релевантных товаров}}.$$

Нормализованный дисконтированный совокупный выигрыш (normalized discounted cumulative gain – NDGC) измеряет качество системы ранжирования рекомендаций с учетом релевантности элементов, где релевантность обычно представлена оценкой или бинарным значением релевантности (релевантный или нет).

```
def get_user_items(edge_index):
    user_items = dict()
    for i in range(edge_index.shape[1]):
        user = edge_index[0][i].item()
        item = edge_index[1][i].item()
        if user not in user_items:
            user_items[user] = []
        user_items[user].append(item)
    return user_items

def compute_recall_at_k(items_ground_truth, items_predicted):
    num_correct_pred = np.sum(items_predicted, axis=1)
    num_total_pred = np.array(
        [len(items_ground_truth[i]) for i in range(len(items_ground_truth))]
    )
```

```
    recall = np.mean(num_correct_pred / num_total_pred)

    return recall

def compute_ndcg_at_k(items_ground_truth, items_predicted):
    test_matrix = np.zeros((len(items_predicted), K))

    for i, items in enumerate(items_ground_truth):
        length = min(len(items), K)
        test_matrix[i, :length] = 1

    max_r = test_matrix
    idcg = np.sum(max_r * 1. / np.log2(np.arange(2, K + 2)), axis=1)
    dcg = items_predicted * (1. / np.log2(np.arange(2, K + 2)))
    dcg = np.sum(dcg, axis=1)
    idcg[idcg == 0.] = 1.
    ndcg = dcg / idcg
    ndcg[np.isnan(ndcg)] = 0.

    return np.mean(ndcg)

# функция-обертка для получения метрик качества
def get_metrics(model, edge_index, exclude_edge_indices):

    ratings = torch.matmul(model.emb_users.weight, model.emb_items.weight.T)

    for exclude_edge_index in exclude_edge_indices:
        user_pos_items = get_user_items(exclude_edge_index)
        exclude_users = []
        exclude_items = []
        for user, items in user_pos_items.items():
            exclude_users.extend([user] * len(items))
            exclude_items.extend(items)
        ratings[exclude_users, exclude_items] = -1024

    # получаем топ-k рекомендованных товаров для каждого пользователя
    _, top_K_items = torch.topk(ratings, k=K)

    # получаем всех уникальных пользователей в оцененном нами наборе
    users = edge_index[0].unique()

    test_user_pos_items = get_user_items(edge_index)

    # преобразовываем словарь положительных товаров
    # для тестового пользователя в список
    test_user_pos_items_list = [
        test_user_pos_items[user.item()] for user in users
    ]

    # определяем правильность топ-k прогнозов
    items_predicted = []
    for user in users:
        ground_truth_items = test_user_pos_items[user.item()]
        label = list(map(lambda x: x in ground_truth_items, top_K_items[user]))
        items_predicted.append(label)

    recall = compute_recall_at_k(test_user_pos_items_list, items_predicted)
```

```
    ndcg = compute_ndcg_at_k(test_user_pos_items_list, items_predicted)

    return recall, ndcg

# функция-обертка для оценки качества модели
def test(model, edge_index, exclude_edge_indices):
    emb_users_final, emb_users, emb_items_final, emb_items = model.forward(
        edge_index
    )
    user_indices, pos_item_indices, neg_item_indices = (
        structured_negative_sampling(edge_index,
                                     contains_neg_self_loops=False)
    )

    emb_users_final, emb_users = (
        emb_users_final[user_indices],
        emb_users[user_indices]
    )

    emb_pos_items_final, emb_pos_items = (
        emb_items_final[pos_item_indices],
        emb_items[pos_item_indices]
    )
    emb_neg_items_final, emb_neg_items = (
        emb_items_final[neg_item_indices],
        emb_items[neg_item_indices]
    )

    loss = bpr_loss(
        emb_users_final,
        emb_users,
        emb_pos_items_final,
        emb_pos_items,
        emb_neg_items_final,
        emb_neg_items
    ).item()

    recall, ndcg = get_metrics(model, edge_index, exclude_edge_indices)

    return loss, recall, ndcg
```

Теперь мы можем написать цикл обучения и начать обучение модели LightGCN.

1. Определяем следующие константы. Они могут быть настроены как гиперпараметры для улучшения качества модели:

```
K = 20
LAMBDA = 1e-6
BATCH_SIZE = 1024
```

2. Постараемся воспользоваться GPU, если будет такая возможность. В противном случае мы используем CPU. Модель и данные перемещаются на это устройство:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model = LightGCN(num_users, num_items)
model = model.to(device)
edge_index = edge_index.to(device)
train_edge_index = train_edge_index.to(device)
val_edge_index = val_edge_index.to(device)
```

3. Мы задаем оптимизатор Adam с темпом обучения 0.001:

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

4. Запустим цикл обучения. Сначала вычисляем `num_batch`, количество батчей `BATCH_SIZE` в одной эпохе. Затем создадим два цикла: один из 31 эпохи, а второй – длиной `num_batch`:

```
n_batch = int(len(train_index)/BATCH_SIZE)

for epoch in range(31):
    model.train()

    for _ in range(n_batch):
```

5. Модель запускается на обучающих данных и возвращает исходные и итоговые эмбеддинги пользователей и товаров:

```
        optimizer.zero_grad()

        emb_users_final, emb_users, emb_items_final, emb_items = model.forward(
            train_edge_index
        )
```

6. Затем обучающие данные отбираются в мини-батчи с помощью функции `sample_mini_batch()`, которая возвращает индексы отобранных пользователей, положительных и отрицательных товаров:

```
        user_indices, pos_item_indices, neg_item_indices = sample_mini_batch(
            train_edge_index
        )
```

7. Затем извлекаем эмбеддинги для отобранных пользователей, положительных и отрицательных товаров:

```
        emb_users_final, emb_users = (
            emb_users_final[user_indices],
            emb_users[user_indices]
        )
        emb_pos_items_final, emb_pos_items = (
            emb_items_final[pos_item_indices],
            emb_items[pos_item_indices]
        )
        emb_neg_items_final, emb_neg_items = (
```

```
        emb_items_final[neg_item_indices],
        emb_items[neg_item_indices]
    )
```

8. Затем вычисляем функцию потерь с помощью функции `bpr_loss()`:

```
train_loss = bpr_loss(
    emb_users_final,
    emb_users,
    emb_pos_items_final,
    emb_pos_items,
    emb_neg_items_final,
    emb_neg_items
)
```

9. Затем оптимизатор используется для выполнения обратного прохода и обновления параметров модели:

```
train_loss.backward()
optimizer.step()
```

10. Качество модели оценивается каждые 5 эпох на проверочном наборе с помощью функции `test()`. Метрики оценки выводятся на печать:

```
if epoch % 5 == 0:
    model.eval()
    val_loss, recall, ndcg = test(
        model, val_edge_index, [train_edge_index]
    )
    print(
        f"Эпоха {epoch} \n| Функция потерь - обуч. набор: "
        f"{train_loss.item():.5f} | "
        f"Функция потерь - проверочн. набор: {val_loss:.5f} \n| "
        f"Recall@{K} - проверочн. набор: {recall:.5f} | "
        f"Ndcg@{K} - проверочн. набор: {ndcg:.5f}"
    )
```

11. В итоге получаем следующий вывод:

```
Эпоха 0
| Функция потерь - обуч. набор: -0.69345 | Функция потерь - проверочн. набор: -0.69255
| Recall@20 - проверочн. набор: 0.00985 | Ndcg@20 - проверочн. набор: 0.00567
Эпоха 5
| Функция потерь - обуч. набор: -0.73150 | Функция потерь - проверочн. набор: -0.63679
| Recall@20 - проверочн. набор: 0.01870 | Ndcg@20 - проверочн. набор: 0.00921
Эпоха 10
| Функция потерь - обуч. набор: -0.96728 | Функция потерь - проверочн. набор: -0.42531
| Recall@20 - проверочн. набор: 0.01859 | Ndcg@20 - проверочн. набор: 0.00919
Эпоха 15
| Функция потерь - обуч. набор: -1.35675 | Функция потерь - проверочн. набор: -0.02465
| Recall@20 - проверочн. набор: 0.01889 | Ndcg@20 - проверочн. набор: 0.00931
Эпоха 20
| Функция потерь - обуч. набор: -2.22876 | Функция потерь - проверочн. набор: 0.52749
```

```
| Recall@20 - проверочн. набор: 0.01863 | Ndcg@20 - проверочн. набор: 0.00918
Эпоха 25
| Функция потерь - обуч. набор: -3.23828 | Функция потерь - проверочн. набор: 1.19839
| Recall@20 - проверочн. набор: 0.01835 | Ndcg@20 - проверочн. набор: 0.00915
Эпоха 30
| Функция потерь - обуч. набор: -4.22176 | Функция потерь - проверочн. набор: 1.97920
| Recall@20 - проверочн. набор: 0.01791 | Ndcg@20 - проверочн. набор: 0.00908
```

12. Затем оцениваем качество модели на тестовом наборе:

```
test_loss, test_recall, test_ndcg = test(
    model, test_edge_index.to(device),
    [train_edge_index, val_edge_index]
)

print(
    f"Функция потерь - тест. набор: {test_loss:.5f} | "
    f"Recall@{K} - тест. набор: {test_recall:.5f} | "
    f"Ndcg@{K} - тест. набор: {test_ndcg:.5f}"
)
```

```
Функция потерь - тест. набор: 1.94753 | Recall@20 - тест. набор: 0.01716 | Ndcg@20 -
тест. набор: 0.00908
```

Мы получили значение recall@20, равное 0.01936, и значение ndcg@20, равное 0.01119, что близко к результатам, полученным авторами LightGCN на других наборах данных.

Теперь, когда модель обучена, мы хотим получить рекомендации для конкретного пользователя. Функция рекомендации, которую мы хотим написать, состоит из двух компонент.

1. Во-первых, мы хотим получить список книг, которые понравились пользователю. Это поможет нам контекстуализировать рекомендации.
2. Во-вторых, мы хотим сформировать список рекомендаций. Эти рекомендации не могут быть книгами, которые пользователь уже оценил (это не может быть положительный товар).

```
bookid_title = pd.Series(books['Book-Title'].values,
                         index=books.ISBN).to_dict()
bookid_author = pd.Series(books['Book-Author'].values,
                          index=books.ISBN).to_dict()
user_pos_items = get_user_items(edge_index)
```

Давайте напишем эту функцию шаг за шагом.

1. Пишем функцию `recommend()`, которая принимает два аргумента – `user_id` (идентификатор пользователя) и `num_recs` (количество рекомендаций, которое мы хотим сгенерировать):

```
from PIL import Image
import requests

def recommend(user_id, num_recs):
```

2. Создаем переменную `user`, отыскивая идентификатор пользователя в словаре `user_mapping`, который сопоставляет идентификаторы пользователей с целочисленными индексами:

```
user = user_mapping[user_id]
```

3. Получаем вектор размерности `dim_h`, выученный моделью LightGCN для этого конкретного пользователя:

```
emb_user = model.emb_users.weight[user]
```

4. Его можно использовать для вычисления соответствующих рейтингов. Как было показано ранее, мы используем скалярное произведение эмбеддингов для всех товаров, хранящихся в атрибуте `emb_items` сети LightGCN, и переменной `emb_user`:

```
ratings = model.emb_items.weight @ emb_user
```

5. Применяем к тензору `ratings` функцию `topk()`, которая возвращает 100 лучших значений (оценок, вычисленных моделью) и соответствующие им индексы:

```
values, indices = torch.topk(ratings, k=100)
```

6. Получаем список любимых книг этого пользователя. Мы создаем новый список индексов, фильтруя список `indices` так, чтобы в него входили только те индексы, которые присутствуют в словаре `user_items` для данного пользователя. Другими словами, мы сохраняем только те книги, которые этот пользователь оценил. Затем берем срез списка, чтобы сохранить первые `num_recs` элементов:

```
ids = [index.cpu().item() for index in indices
       if index in user_pos_items[user]][:num_recs]
```

7. Преобразуем эти идентификаторы книг в ISBN:

```
item_isbns = [
    list(item_mapping.keys())[list(item_mapping.values()).index(book)]
    for book in ids
]
```

8. Теперь воспользуемся этими ISBN для получения дополнительной информации о книгах. Здесь нам нужно получить названия книг и авторов, чтобы можно было их распечатать:

```
titles = [bookid_title[id] for id in item_isbns]
authors = [bookid_author[id] for id in item_isbns]
```

9. Распечатаем эту информацию следующим образом:

```
print(f' Любимые книги для пользователя n°{user_id}:')
for i in range(len(item_isbns)):
    print(f'- {titles[i]}, by {authors[i]}')
```

10. Мы повторяем этот процесс, но уже с идентификаторами книг, которые не были оценены пользователем (`not in user_pos_items[user]`):

```
    ids = [index.cpu().item() for index in indices
           if index not in user_pos_items[user]][:num_recs]
    item_isbns = [
        list(item_mapping.keys())[list(item_mapping.values()).index(book)]
        for book in ids
    ]
    titles = [bookid_title[id] for id in item_isbns]
    authors = [bookid_author[id] for id in item_isbns]

    print(f'\n Рекомендованные книги для пользователя n°{user_id}')
    for i in range(num_recs):
        print(f'- {titles[i]}, by {authors[i]}')

    user_agent_value = ('Mozilla/5.0 (Windows NT 10.0; Win64; x64) '
                        'AppleWebKit/537.36 (KHTML, like Gecko) '
                        'Chrome/58.0.3029.110 Safari/537.3')

    headers = {'User-Agent': user_agent_value}
    fig, axs = plt.subplots(1, num_recs, figsize=(20,6))
    fig.patch.set_alpha(0)
    for i, title in enumerate(titles):
        url = books.loc[books['Book-Title'] == title]['Image-URL-L'][:1].values[0]
        img = Image.open(requests.get(url, stream=True, headers=headers).raw)
        rating = df.loc[
            df['ISBN'] == books.loc[books['Book-Title'] == title]['ISBN'][:1].
values[0]
        ]['Book-Rating'].mean()
        axs[i].axis("off")
        axs[i].imshow(img)
        axs[i].set_title(f'{rating:.1f}/10', y=-0.1, fontsize=18)
```

11. Давайте получим 5 рекомендаций для пользователя в нашей базе данных. Возьмем пользователя 277427:

```
recommend(277427, 5)
```

12. Вот вывод, который мы получаем:

```
Любимые книги для пользователя n°277427:
- The Da Vinci Code, by Dan Brown
- One for the Money (Stephanie Plum Novels (Paperback)), by Janet Evanovich
- Into the Wild, by Jon Krakauer

Рекомендованные книги для пользователя n°277427
- The Lovely Bones: A Novel, by Alice Sebold
- The Red Tent (Bestselling Backlist), by Anita Diamant
- Harry Potter and the Sorcerer's Stone (Harry Potter (Paperback)), by J. K. Rowling
- To Kill a Mockingbird, by Harper Lee
- Life of Pi, by Yann Martel
```

Теперь мы можем генерировать рекомендации для любого пользователя из исходного датафрейма `df`. Вы можете выбрать других пользователей и посмотреть, как это изменит рекомендации.

Выводы

В этой главе мы подробно рассмотрели использование LightGCN для выдачи рекомендаций книг. Мы использовали набор данных `Book-Crossing`, предварительно обработали его, чтобы сформировать двудольный граф, и реализовали модель LightGCN с функцией потерь BPR. Мы обучили модель и оценили ее с помощью метрик `recall@20` и `ndcg@20`. Мы продемонстрировали эффективность модели на примере генерации рекомендаций для данного пользователя.

В целом эта глава позволила получить ценные сведения об использовании моделей LightGCN в задачах выдачи рекомендаций. Это современная архитектура, которая работает лучше, чем более сложные модели. Вы можете расширить этот проект, попробовав другие методы, которые мы обсуждали в предыдущих главах, такие как матричная факторизация и `node2vec`.

Дополнительное чтение

[1] C.-N. Ziegler, S. M. McNee, J. A. Konstan, and G. Lausen, *Improving Recommendation Lists through Topic Diversification, in Proceedings of the 14th International Conference on World Wide Web*, 2005, pp. 22–32. doi: 10.1145/1060745.1060754. Доступ по ссылке https://dl.acm.org/doi/10.1145/ 1060745.1060754.

[2] D. Li, P. Maldonado, A. Sbaih, *Recommender Systems with GNNs in PyG, Stanford CS224W GraphML Tutorials*, 2022. Доступ по ссылке https://medium.com/stanford-cs224w/recommender-systems-with-gnns-in-pyg-d8301178e377.

[3] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang, *LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation*. arXiv,

2020. doi: 10.48550/ARXIV.2002.02126. Доступ по ссылке https://arxiv.org/abs/2002.02126.

[4] H. Hotta and A. Zhou, *LightGCN with PyTorch Geometric. Stanford CS224W GraphML Tutorials*, 2022. Доступ по ссылке https://medium.com/stanford-cs224w/lightgcn-with-pytorch-geometric-91bab836471e.

Глава 17

Обнаружение аномалий с помощью гетерогенных графовых нейронных сетей

В машинном обучении обнаружение аномалий – это популярная задача, целью которой является выявление паттернов или наблюдений в данных, которые отклоняются от ожидаемого поведения. Это фундаментальная задача, возникающая во многих прикладных областях, таких как обнаружение мошенничества в финансовых операциях, выявление бракованной продукции в производственном процессе или обнаружение кибератак в компьютерной сети.

Мы можем обучить GNN, чтобы она выучила нормальный режим работы компьютерной сети, а затем выявить узлы или паттерны, которые отклоняются от этого нормального режима. Действительно, способность графовых нейронных сетей понимать сложные взаимосвязи делает их особенно подходящими для обнаружения слабых сигналов. Кроме того, GNN можно масштабировать на большие наборы данных, что делает их эффективным инструментом для обработки больших объемов данных.

В этой главе мы применим GNN для обнаружения аномалий в компьютерных сетях. Во-первых, представим набор данных CIDDS-001, который содержит атаки и нормальный трафик компьютерной сети. Затем обработаем набор данных, подготовив его для передачи в GNN. Затем перейдем к реализации гетерогенной GNN для работы с различными типами узлов и ребер. Наконец, мы обучим сеть, используя обработанный набор данных, и оценим результаты, чтобы увидеть, насколько хорошо она обнаруживает аномалии в сетевом трафике.

К концу этой главы вы будете знать, как реализовать GNN для обнаружения вторжений. Кроме того, вы узнаете, как создавать релевантные признаки для обнаружения атак и обрабатывать их с последующей передачей в GNN.

Наконец, вы узнаете, как реализовать и оценить гетерогенную GNN для обнаружения редких атак.

В этой главе мы рассмотрим следующие основные темы:

- «Исследование набора данных CIDDS-001»,
- «Предварительная обработка набора данных CIDDS-001»,
- «Реализация гетерогенной GNN».

Технические требования

Все примеры кода из этой главы можно найти на GitHub по адресу https://github.com/Gewissta/GNN/blob/main/Chapter17/Chapter17.ipynb.

Исследование набора данных CIDDS-001

В этом разделе мы проведем разведочный анализ набора данных и попробуем получить более подробную информацию о важности признаков и полезности преобразований.

Набор данных CIDDS-001 [1] предназначен для обучения и оценки качества систем обнаружения вторжений в сеть, использующих анализ аномалий. Он содержит реалистичный трафик, включающий современные атаки, для оценки этих систем. Его создали путем сбора и разметки 8 451 520 потоков трафика в виртуальной среде с помощью OpenStack. Если говорить более точно, каждая строка соответствует NetFlow-соединению, описывая статистику трафика **интернет-протокола** (Internet Protocol – IP), например количество обмененных байтов.

На следующем рисунке представлен обзор симулированной сетевой среды в `CIDDS-001`.

Мы видим четыре разные подсети (разработчик, офис, управление и сервер) с соответствующими диапазонами IP-адресов. Все эти подсети связаны с одним сервером, подключенным к интернету через файервол. Также присутствует внешний сервер, который предоставляет два сервиса: сервис синхронизации файлов и веб-сервер. Наконец, за пределами локальной сети представлены атакующие. Соединения в `CIDDS-001` были собраны с локальных и внешних серверов. Цель этого набора данных – правильно классифицировать эти соединения на пять категорий: нормальные соединения (без атак), брутфорс-атаки (атаки полным перебором), DoS-атаки (атаки на систему с целью довести ее до отказа), сканирование с помощью команды ping (сканирование с целью определить доступность узлов и устройств в сети) и сканирование портов (сканирование для определения открытых портов).

Давайте скачаем набор данных CIDDS-001, прочитаем в датафрейм и изучим его входные характеристики.

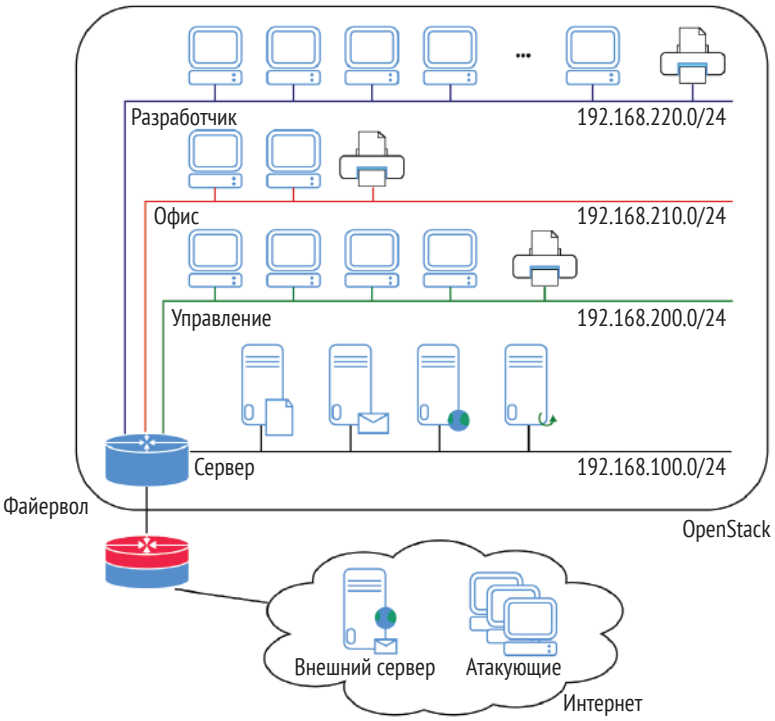


Рис. 17.1 ❖ Обзор виртуальной сети, симулированной с помощью набора данных CIDDS-001

1. Скачиваем CIDDS-001:

```
from io import BytesIO
from urllib.request import urlopen
from zipfile import ZipFile

# скачивание может быть очень долгим!
url = 'https://www.hs-coburg.de/fileadmin/hscoburg/WISENT-CIDDS-001.zip'
with urlopen(url) as zurl:
    with ZipFile(BytesIO(zurl.read())) as zfile:
        zfile.extractall('.')
```

2. Импортируем необходимые библиотеки:

```
import numpy as np
import pandas as pd
pd.set_option('display.max_columns', 100)
import itertools
import matplotlib.pyplot as plt
```

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PowerTransformer
from sklearn.metrics import (f1_score,
                             classification_report,
                             confusion_matrix)

from torch_geometric.loader import DataLoader
from torch_geometric.data import HeteroData
from torch.nn import functional as F
from torch.optim import Adam
from torch import nn
```

3. Считываем набор данных в датафрейм pandas:

```
df = pd.read_csv("CIDDS-001/traffic/OpenStack/CIDDS-001-internal-week1.csv")
df
```

	Date first seen	Duration	Proto	Src IP Addr	Src Pt	Dst IP Addr	Dst Pt	Packets	Bytes	Flows	Flags	Tos	class	attackType	attackID
0	2017-03-15 00:01:16.632	0.000	TCP	192.168.100.5	445	192.168.220.16	58844.0	1	108	1	.AP...	0	normal	---	---
1	2017-03-15 00:01:16.552	0.000	TCP	192.168.100.5	445	192.168.220.15	48888.0	1	108	1	.AP...	0	normal	---	---
2	2017-03-15 00:01:16.551	0.004	TCP	192.168.220.15	48888	192.168.100.5	445.0	2	174	1	.AP...	0	normal	---	---
3	2017-03-15 00:01:16.631	0.004	TCP	192.168.220.16	58844	192.168.100.5	445.0	2	174	1	.AP...	0	normal	---	---
4	2017-03-15 00:01:16.552	0.000	TCP	192.168.100.5	445	192.168.220.15	48888.0	1	108	1	.AP...	0	normal	---	---
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
8451515	2017-03-21 23:59:56.083	0.248	TCP	192.168.200.8	62605	EXT_SERVER	8082.0	2	319	1	.AP...	0	normal	---	---
8451516	2017-03-21 23:59:57.037	0.000	TCP	10179_174	443	192.168.210.5	51433.0	1	54	1	.A....	32	normal	---	---
8451517	2017-03-21 23:59:56.920	0.000	TCP	192.168.210.5	51433	10179_174	443.0	1	55	1	.A....	0	normal	---	---
8451518	2017-03-21 23:59:58.299	0.000	TCP	192.168.100.5	445	192.168.220.6	56281.0	1	108	1	.AP...	0	normal	---	---
8451519	2017-03-21 23:59:58.298	0.002	TCP	192.168.220.6	56281	192.168.100.5	445.0	2	174	1	.AP...	0	normal	---	---

8451520 rows × 16 columns

Есть несколько интересных признаков, которые мы можем использовать для нашей модели.

- Дата первого появления – это метка времени, из которой мы можем извлечь информацию о дне недели и времени суток. В целом сетевой трафик является сезонным, и соединения, которые происходят ночью или в необычные дни, вызывают подозрения.
- IP-адреса (например, `192.168.100.5`) сложны для обработки, потому что не являются количественными значениями и подчиняются сложному набору правил. Мы можем разбить их на несколько категорий (выполнить биннинг), так как знаем, как настроена наша локальная сеть. Другое популярное и более обобщенное решение – преобразовать их в двоичное представление («192» превращается в «11000000»).
- Длительность, количество пакетов и количество байтов – это признаки, которые обычно имеют тяжелые распределения. Поэтому они потребуют специальной обработки.

Давайте проверим последний пункт и внимательно рассмотрим распределение атак в этом наборе данных.

1. Начнем с удаления признаков, которые мы не будем рассматривать в этом проекте: порты, количество потоков, тип сервиса, класс, идентификатор атаки и описание атаки:

```
df = df.drop(columns=['Src Pt', 'Dst Pt', 'Flows', 'Tos',
                      'class', 'attackID', 'attackDescription'])
```

2. Мы переименуем класс `'---'` переменной *attackType* в `'benign'`, а для признака `date first seen` зададим тип данных `timestamp`:

```
df['attackType'] = df['attackType'].replace('---', 'benign')
df['Date first seen'] = pd.to_datetime(df['Date first seen'])
df
```

	Date first seen	Duration	Proto	Src IP Addr	Dst IP Addr	Packets	Bytes	Flags	attackType
0	2017-03-15 00:01:16.632	0.000	TCP	192.168.100.5	192.168.220.16	1	108	.AP...	benign
1	2017-03-15 00:01:16.552	0.000	TCP	192.168.100.5	192.168.220.15	1	108	.AP...	benign
2	2017-03-15 00:01:16.551	0.004	TCP	192.168.220.15	192.168.100.5	2	174	.AP...	benign
3	2017-03-15 00:01:16.631	0.004	TCP	192.168.220.16	192.168.100.5	2	174	.AP...	benign
4	2017-03-15 00:01:16.552	0.000	TCP	192.168.100.5	192.168.220.15	1	108	.AP...	benign
...	...	...	...	...	...	...	...	...	...
8451515	2017-03-21 23:59:56.083	0.248	TCP	192.168.200.8	EXT_SERVER	2	319	.AP...	benign
8451516	2017-03-21 23:59:57.037	0.000	TCP	10179_174	192.168.210.5	1	54	.A....	benign
8451517	2017-03-21 23:59:56.920	0.000	TCP	192.168.210.5	10179_174	1	55	.A....	benign
8451518	2017-03-21 23:59:58.299	0.000	TCP	192.168.100.5	192.168.220.6	1	108	.AP...	benign
8451519	2017-03-21 23:59:58.298	0.002	TCP	192.168.220.6	192.168.100.5	2	174	.AP...	benign

8451520 rows × 9 columns

3. Подсчитаем частоты классов и построим круговую диаграмму с тремя наиболее представленными классами (два остальных класса составляют менее 0.1 %):

```
count_labels = df['attackType'].value_counts() / len(df) * 100
print(count_labels)
plt.pie(count_labels[:3],
        labels=df['attackType'].unique()[:3],
        autopct='%.0f%%')
plt.show()
```

```
benign        82.954273
dos           14.815406
portScan       2.171337
pingScan       0.039744
bruteForce     0.019239
Name: attackType, dtype: float64
```

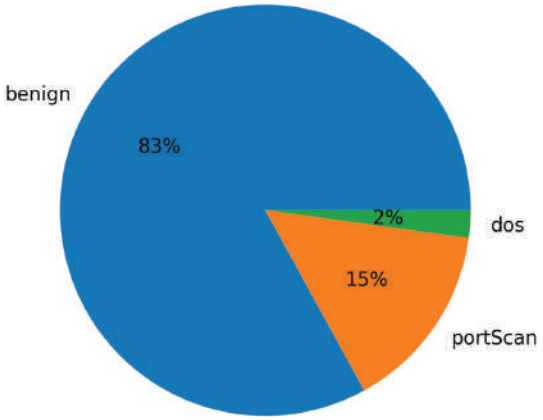


Рис. 17.2 ❖ Доля каждого класса в наборе данных CIDDS-001

Как видите, нормальный трафик составляет подавляющее большинство в этом наборе данных. Напротив, брутфорс-атаки и пинг-сканирование представлены очень малым количеством наблюдений. Такая несбалансированность данных может негативно сказаться на качестве модели при работе с редкими классами.

4. Наконец, мы можем визуализировать распределения длительности, количества пакетов и количество байт. Это позволяет нам понять, действительно ли они нуждаются в специальных преобразованиях:

```
fig, ((ax1, ax2, ax3)) = plt.subplots(1, 3, figsize=(15,5))
df['Duration'].hist(ax=ax1)
ax1.set_xlabel("Duration")
df['Packets'].hist(ax=ax2)
ax2.set_xlabel("Number of packets")
pd.to_numeric(df['Bytes'], errors='coerce').hist(ax=ax3)
ax3.set_xlabel("Number of bytes")
plt.show()
```

В итоге получаем графики, представленные на рис. 17.3.

Мы видим, что большинство значений близко к нулю и есть длинный хвост редких значений, вытянутый вдоль оси x. Воспользуемся преобразованием Бокса-Кокса, чтобы сделать эти распределения признаков более похожими на гауссово распределение, что поможет модели в процессе обучения. Теперь, когда мы изучили основные характеристики набора данных `CIDDS-001`, можем перейти к этапу предварительной обработки.

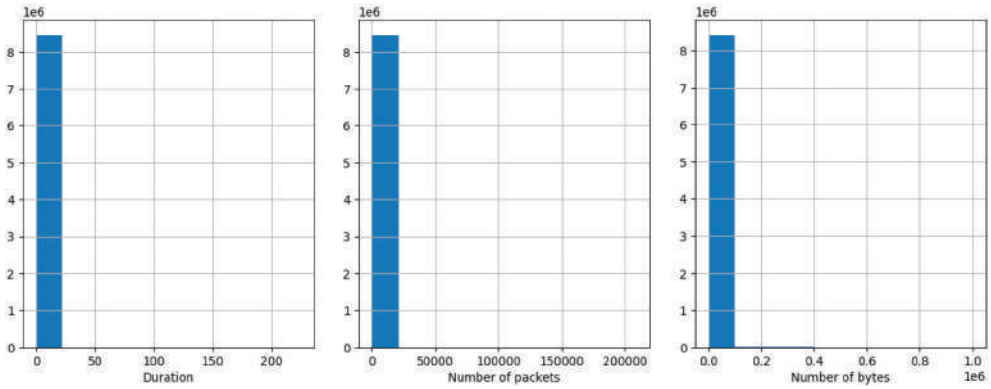


Рис. 17.3 ❖ Распределения признаков Duration, Number of packets и Number of bytes

Предварительная обработка набора данных CIDDS-001

В предыдущем разделе мы определили некоторые проблемы с набором данных, которые необходимо решить, чтобы повысить качество нашей модели.

Набор данных `CIDDS-001` включает в себя различные типы данных: у нас есть количественные признаки типа длительности, категориальные признаки, например тип протокола (TCP, UDP, ICMP и IGMP), и некоторые другие признаки, такие как временные метки или IP-адреса. Далее мы выберем способ представления этих типов данных, основываясь на информации из предыдущего раздела и экспертных знаниях.

1. Во-первых, мы извлекаем из временной метки дни недели и применяем к ним one-hot-кодирование. Затем переименовываем получившиеся столбцы, чтобы сделать их более читаемыми:

```
df['weekday'] = df['Date first seen'].dt.weekday
df = pd.get_dummies(df, columns=['weekday']).rename(
    columns={'weekday_0': 'Monday',
             'weekday_1': 'Tuesday',
             'weekday_2': 'Wednesday',
             'weekday_3': 'Thursday',
             'weekday_4': 'Friday',
             'weekday_5': 'Saturday',
             'weekday_6': 'Sunday'}
)
```

2. Еще один важный тип информации, которую мы можем получить с помощью временных меток, – это время. Мы также нормализуем его в диапазоне от 0 до 1:

```
df['daytime'] = (df['Date first seen'].dt.second +
                 df['Date first seen'].dt.minute * 60 +
                 df['Date first seen'].dt.hour * 60 * 60) / (24 * 60 * 60)
```

3. Мы еще не упомянули о флагах TCP. Каждый флаг указывает на определенное состояние TCP-соединения. Например, `F` или `FIN` означает, что пир TCP-соединения завершил отправку данных. Мы можем извлечь каждый флаг и применить к ним one-hot-кодирование следующим образом:

```
def one_hot_flags(input):
    return [1 if char1 == char2 else 0 for char1, char2
            in zip('APRSF', input[1:])]

df = df.reset_index(drop=True)
ohe_flags = one_hot_flags(df['Flags'].to_numpy())
ohe_flags = df['Flags'].apply(one_hot_flags).to_list()
df[['ACK', 'PSH', 'RST', 'SYN', 'FIN']] = pd.DataFrame(
    ohe_flags, columns=['ACK', 'PSH', 'RST', 'SYN', 'FIN']
)
df = df.drop(columns=['Date first seen', 'Flags'])
df
```

	Duration	Proto	Src IP Addr	Dst IP Addr	Packets	Bytes	attackType	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
0	0.000	TCP	192.168.100.5	192.168.220.16	1	108	benign	0	0	1	0	0	0	0
1	0.000	TCP	192.168.100.5	192.168.220.15	1	108	benign	0	0	1	0	0	0	0
2	0.004	TCP	192.168.220.15	192.168.100.5	2	174	benign	0	0	1	0	0	0	0
3	0.004	TCP	192.168.220.16	192.168.100.5	2	174	benign	0	0	1	0	0	0	0
4	0.000	TCP	192.168.100.5	192.168.220.15	1	108	benign	0	0	1	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
8451515	0.248	TCP	192.168.200.8	EXT_SERVER	2	319	benign	0	1	0	0	0	0	0
8451516	0.000	TCP	10179_174	192.168.210.5	1	54	benign	0	1	0	0	0	0	0
8451517	0.000	TCP	192.168.210.5	10179_174	1	55	benign	0	1	0	0	0	0	0
8451518	0.000	TCP	192.168.100.5	192.168.220.6	1	108	benign	0	1	0	0	0	0	0
8451519	0.002	TCP	192.168.220.6	192.168.100.5	2	174	benign	0	1	0	0	0	0	0

daytime	ACK	PSH	RST	SYN	FIN
0.000880	1	1	0	0	0
0.000880	1	1	0	0	0
0.000880	1	1	0	0	0
0.000880	1	1	0	0	0
0.000880	1	1	0	0	0
...	...	...	...	...	...
0.999954	1	1	0	0	0
0.999965	1	0	0	0	0
0.999954	1	0	0	0	0
0.999977	1	1	0	0	0
0.999977	1	1	0	0	0

4. Давайте теперь обработаем IP-адреса. В этом примере мы будем использовать двоичное кодирование. Вместо того чтобы использовать 32 бита для кодирования полного IPv4-адреса, будем оставлять только последние 16 бит, которые здесь являются наиболее важными. Действительно, первые 16 бит соответствуют либо 192.168, если хост принадлежит внутренней сети, либо другому значению, если он является внешним:

```
temp = pd.DataFrame()
temp['SrcIP'] = df['Src IP Addr'].astype(str)
temp['SrcIP'][~temp['SrcIP'].str.contains(
    '\d{1,3}\.', regex=True)] = '0.0.0.0'
temp = temp['SrcIP'].str.split('.', expand=True).rename(
    columns = {2: 'ipsrc3', 3: 'ipsrc4'}).astype(int)[['ipsrc3', 'ipsrc4']]
temp['ipsrc'] = temp['ipsrc3'].apply(
    lambda x: format(x, "b").zfill(8)) + temp['ipsrc4'].apply(
    lambda x: format(x, "b").zfill(8))
df = df.join(temp['ipsrc'].str.split('', expand=True)
             .drop(columns=[0, 17])
             .rename(columns=dict(enumerate(
                 [f'ipsrc_{i}' for i in range(17)])))
             .astype('int32'))
df.head(5)
```

	Duration	Proto	Src IP Addr	Dst IP Addr	Packets	Bytes	attackType	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday	daytime
0	0.000	TCP	192.168.100.5	192.168.220.16	1	108	benign	0	0	1	0	0	0	0	0.00088
1	0.000	TCP	192.168.100.5	192.168.220.15	1	108	benign	0	0	1	0	0	0	0	0.00088
2	0.004	TCP	192.168.220.15	192.168.100.5	2	174	benign	0	0	1	0	0	0	0	0.00088
3	0.004	TCP	192.168.220.16	192.168.100.5	2	174	benign	0	0	1	0	0	0	0	0.00088
4	0.000	TCP	192.168.100.5	192.168.220.15	1	108	benign	0	0	1	0	0	0	0	0.00088

ACK	PSH	RST	SYN	FIN	ipsrc_1	ipsrc_2	ipsrc_3	ipsrc_4	ipsrc_5	ipsrc_6	ipsrc_7	ipsrc_8	ipsrc_9
1	1	0	0	0	0	1	1	0	0	1	0	0	0
1	1	0	0	0	0	1	1	0	0	1	0	0	0
1	1	0	0	0	1	1	0	1	1	1	0	0	0
1	1	0	0	0	1	1	0	1	1	1	0	0	0
1	1	0	0	0	0	1	1	0	0	1	0	0	0

ipsrc_10	ipsrc_11	ipsrc_12	ipsrc_13	ipsrc_14	ipsrc_15	ipsrc_16
0	0	0	0	1	0	1
0	0	0	0	1	0	1
0	0	0	1	1	1	1
0	0	1	0	0	0	0
0	0	0	0	1	0	1

5. Повторяем этот процесс для IP-адресов назначения:

```
temp = pd.DataFrame()
temp['DstIP'] = df['Dst IP Addr'].astype(str)
temp['DstIP'][~temp['DstIP'].str.contains(
    '\d{1,3}\.', regex=True)] = '0.0.0.0'
temp = temp['DstIP'].str.split('.', expand=True).rename(
```

```
    columns = {2: 'ipdst3', 3: 'ipdst4'}).astype(int)[['ipdst3', 'ipdst4']]
temp['ipdst'] = temp['ipdst3'].apply(lambda x: format(x, "b").zfill(8)) \
              + temp['ipdst4'].apply(lambda x: format(x, "b").zfill(8))
df = df.join(temp['ipdst'].str.split('', expand=True)
           .drop(columns=[0, 17])
           .rename(columns=dict(enumerate([f'ipdst_{i}'
                                           for i in range(17)])))
           .astype('int32'))
df.head(5)
```

	Duration	Proto	Src IP Addr	Dst IP Addr	Packets	Bytes	attackType	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday	daytime
0	0.000	TCP	192.168.100.5	192.168.220.16	1	108	benign	0	0	1	0	0	0	0	0.00088
1	0.000	TCP	192.168.100.5	192.168.220.15	1	108	benign	0	0	1	0	0	0	0	0.00088
2	0.004	TCP	192.168.220.15	192.168.100.5	2	174	benign	0	0	1	0	0	0	0	0.00088
3	0.004	TCP	192.168.220.16	192.168.100.5	2	174	benign	0	0	1	0	0	0	0	0.00088
4	0.000	TCP	192.168.100.5	192.168.220.15	1	108	benign	0	0	1	0	0	0	0	0.00088

ACK	PSH	RST	SYN	FIN	ipsrc_1	ipsrc_2	ipsrc_3	ipsrc_4	ipsrc_5	ipsrc_6	ipsrc_7	ipsrc_8	ipsrc_9	ipsrc_10	ipsrc_11	ipsrc_12	ipsrc_13	ipsrc_14
1	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1
1	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1
1	1	0	0	0	1	1	0	1	1	1	0	0	0	0	0	0	1	1
1	1	0	0	0	1	1	0	1	1	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	1

ipsrc_15	ipsrc_16	ipdst_1	ipdst_2	ipdst_3	ipdst_4	ipdst_5	ipdst_6	ipdst_7	ipdst_8	ipdst_9	ipdst_10	ipdst_11	ipdst_12	ipdst_13
0	1	1	1	0	1	1	1	0	0	0	0	0	1	0
0	1	1	1	0	1	1	1	0	0	0	0	0	0	1
1	1	0	1	1	0	0	1	0	0	0	0	0	0	0
0	0	0	1	1	0	0	1	0	0	0	0	0	0	0
0	1	1	1	0	1	1	1	0	0	0	0	0	0	1

ipdst_14	ipdst_15	ipdst_16
0	0	0
1	1	1
1	0	1
1	0	1
1	1	1

6. Существует проблема с признаком *Bytes*: миллионы представлены в виде символа `m`, а не в виде числового значения. Мы можем исправить это, умножив числовую часть этих нечисловых значений на миллион:

```
m_index = df[pd.to_numeric(
    df['Bytes'], errors='coerce'
).isnull() == True].index
df['Bytes'].loc[m_index] = df['Bytes'].loc[m_index].apply(
    lambda x: 10e6 * float(x.strip().split()[0])
)
df['Bytes'] = pd.to_numeric(df['Bytes'],
                            errors='coerce',
                            downcast='integer')
```

7. Последние признаки, которые нам нужно подвергнуть дамми-кодированию, самые простые: это категориальные признаки, такие как типы

протоколов и типы атак. Мы воспользуемся функцией `get_dummies()` библиотеки pandas:

```
df = pd.get_dummies(df, prefix='', prefix_sep='',
                    columns=['Proto', 'attackType'])
df.head(5)
```

	Duration	Src IP Addr	Dst IP Addr	Packets	Bytes	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday	daytime	ACK	PSH	RST	SYN
0	0.000	192.168.100.5	192.168.220.16	1	108	0	0	1	0	0	0	0	0.00088	1	1	0	0
1	0.000	192.168.100.5	192.168.220.15	1	108	0	0	1	0	0	0	0	0.00088	1	1	0	0
2	0.004	192.168.220.15	192.168.100.5	2	174	0	0	1	0	0	0	0	0.00088	1	1	0	0
3	0.004	192.168.220.16	192.168.100.5	2	174	0	0	1	0	0	0	0	0.00088	1	1	0	0
4	0.000	192.168.100.5	192.168.220.15	1	108	0	0	1	0	0	0	0	0.00088	1	1	0	0

FIN	ipsrc_1	ipsrc_2	ipsrc_3	ipsrc_4	ipsrc_5	ipsrc_6	ipsrc_7	ipsrc_8	ipsrc_9	ipsrc_10	ipsrc_11	ipsrc_12	ipsrc_13	ipsrc_14	ipsrc_15	ipsrc_16	ipdst_1
0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	1	1
0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	1	1
0	1	1	0	1	1	1	0	0	0	0	0	0	1	1	1	1	0
0	1	1	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0
0	0	1	1	0	0	1	0	0	0	0	0	0	0	1	0	1	1

ipdst_2	ipdst_3	ipdst_4	ipdst_5	ipdst_6	ipdst_7	ipdst_8	ipdst_9	ipdst_10	ipdst_11	ipdst_12	ipdst_13	ipdst_14	ipdst_15	ipdst_16	ICMP	IGMP	TCP
1	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	1
1	0	1	1	1	0	0	0	0	0	0	1	1	1	1	0	0	1
1	1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1
1	1	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	0	0	0	0	1	1	1	1	0	0	1

UDP	benign	bruteForce	dos	pingScan	portScan
0	1	0	0	0	0
0	1	0	0	0	0
0	1	0	0	0	0
0	1	0	0	0	0
0	1	0	0	0	0

8. Мы создаем разбиение на обучающий, проверочный и тестовый наборы с соотношением 80/10/10:

```
labels = ['benign', 'bruteForce', 'dos', 'pingScan', 'portScan']
df_train, df_test = train_test_split(
    df, random_state=0, test_size=0.2, stratify=df[labels]
)
df_val, df_test = train_test_split(
    df_test, random_state=0, test_size=0.5, stratify=df_test[labels]
)
```

9. Наконец, нам нужно решить проблему асимметричного распределения трех признаков: длительности, количества пакетов и количества байт. Воспользуемся классом `PowerTransformer` библиотеки scikit-learn для изменения их распределений:

```
boxcox = PowerTransformer()
df_train[['Duration', 'Packets', 'Bytes']] = boxcox.fit_transform(
    df_train[['Duration', 'Packets', 'Bytes']]
)
```

```
df_val[['Duration', 'Packets', 'Bytes']] = boxcox.transform(
    df_val[['Duration', 'Packets', 'Bytes']]
)
df_test[['Duration', 'Packets', 'Bytes']] = boxcox.transform(
    df_test[['Duration', 'Packets', 'Bytes']]
)
```

10. Давайте построим график новых распределений и посмотрим, как они изменились:

```
fig, ((ax1, ax2, ax3)) = plt.subplots(1, 3, figsize=(15,5))
df_train['Duration'].hist(ax=ax1)
ax1.set_xlabel("Duration")
df_train['Packets'].hist(ax=ax2)
ax2.set_xlabel("Number of packets")
df_train['Bytes'].hist(ax=ax3)
ax3.set_xlabel("Number of bytes")
plt.show()
```

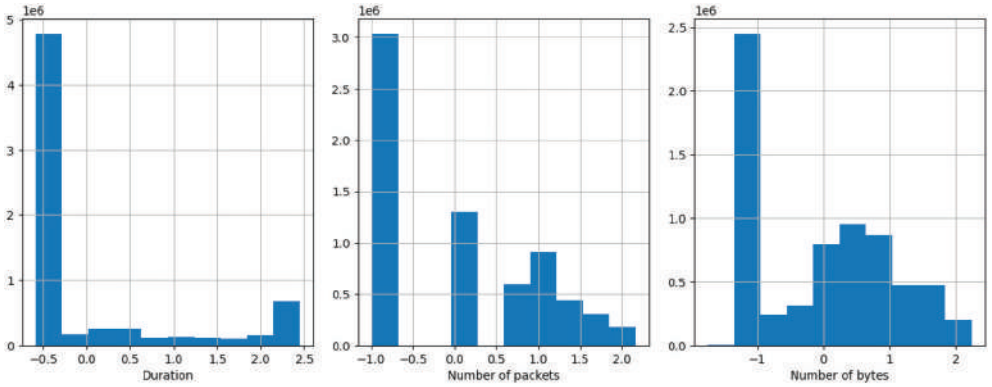


Рис. 17.4 ❖ Новые распределения признаков Duration, Number of packets и Number of bytes

Эти новые распределения не являются гауссовыми, но асимметрия стала менее выраженной, что должно помочь модели.

Обратите внимание, что обработанный нами набор данных является чисто табличным. Нам все равно нужно преобразовать его в графовый набор, прежде чем подавать в GNN. В нашем случае нет очевидного способа преобразовать потоки трафика в узлы. В идеале потоки между одними и теми же компьютерами должны быть связаны. Этого можно достичь с помощью гетерогенного графа с двумя типами узлов:

- **хостами**, которые соответствуют компьютерам и используют IP-адреса в качестве признаков. Если бы у нас было больше информации, мы могли бы добавить другие признаки, связанные с компьютерами, например логи или процент загрузки процессора;

- **потоками**, которые соответствуют соединениям между двумя хостами. Они учитывают все остальные признаки из набора данных. Кроме того, у них есть метка, которую нам нужно предсказать (нормальный или вредоносный поток).

В этом примере потоки являются однонаправленными, поэтому мы также определяем два типа ребер: от хоста к потоку (источник) и от потока к хосту (пункт назначения). Единый граф потребует слишком много памяти, поэтому мы разделим его на подграфы и поместим их в загрузчики данных.

1. Определяем размер батча и признаки, которые мы хотим учитывать для узлов-хостов и узлов-потоков:

```
BATCH_SIZE = 16
features_host = [f'ipsrc_{i}' for i in range(1, 17)] + [
    f'ipdst_{i}' for i in range(1, 17)
]
features_flow = ['daytime', 'Monday', 'Tuesday', 'Wednesday',
                 'Thursday', 'Friday', 'Duration', 'Packets',
                 'Bytes', 'ACK', 'PSH', 'RST', 'SYN', 'FIN',
                 'ICMP ', 'IGMP ', 'TCP  ', 'UDP  ']
```

2. Пишем функцию `get_connections()`, которая получает два индекса ребер из списка IP-адресов, являющихся источниками и пунктами назначения, и словаря `ip_map`:

```
def get_connections(ip_map, src_ip, dst_ip):
```

3. Получаем индексы из IP-адресов (как из источников, так и из пунктов назначения) и состыковываем их:

```
    src1 = [ip_map[ip] for ip in src_ip]
    src2 = [ip_map[ip] for ip in dst_ip]
    src = np.column_stack((src1, src2)).flatten()
```

4. Соединения уникальны, поэтому мы можем легко проиндексировать их с помощью соответствующего диапазона чисел:

```
    dst = list(range(len(src_ip)))
    dst = np.column_stack((dst, dst)).flatten()
```

5. Наконец, возвращаем два следующих индекса ребер:

```
    return (
        torch.Tensor([src, dst]).int(),
        torch.Tensor([dst, src]).int()
    )
```

6. Пишем функцию, которая будет создавать наши загрузчики данных. Она принимает два параметра: созданный нами табличный датафрейм и размер подграфа (1024 узла в данном примере):

```
def create_dataloader(df, subgraph_size=1024):
```

7. Инициализируем список `data` для хранения наших подграфов и подсчитываем количество подграфов, которые нам нужно создать:

```
data = []
n_subgraphs = len(df) // subgraph_size
```

8. Для каждого подграфа мы получаем соответствующие наблюдения в датафрейме, список IP-адресов, являющихся источниками, и список IP-адресов, являющихся пунктами назначения:

```
for i in range(1, n_subgraphs+1):
    subgraph = df[(i-1) * subgraph_size:i * subgraph_size]
    src_ip = subgraph['Src IP Addr'].to_numpy()
    dst_ip = subgraph['Dst IP Addr'].to_numpy()
```

9. Создаем словарь, который сопоставляет IP-адрес с индексом узла:

```
ip_map = {ip:index for index, ip in enumerate(
    np.unique(np.append(src_ip, dst_ip)))}
```

10. Этот словарь поможет нам создать индекс ребер, идущих от хоста к потоку и наоборот. Мы используем функцию `get_connections()`, которую написали ранее.

```
host_to_flow, flow_to_host = get_connections(
    ip_map, src_ip, dst_ip
)
```

11. Используем все собранные данные, чтобы создать гетерогенный граф для каждого подграфа, и добавляем его в список:

```
batch = HeteroData()
batch['host'].x = torch.Tensor(
    subgraph[features_host].to_numpy()).float()
batch['flow'].x = torch.Tensor(
    subgraph[features_flow].to_numpy()).float()
batch['flow'].y = torch.Tensor(
    subgraph[labels].to_numpy()).float()
batch['host','flow'].edge_index = host_to_flow
batch['flow','host'].edge_index = flow_to_host
data.append(batch)
```

12. Наконец, мы возвращаем загрузчик данных с соответствующим размером батча:

```
return DataLoader(data, batch_size=BATCH_SIZE)
```

13. Теперь, когда у нас есть все необходимое, мы можем вызвать функцию `create_dataloader()` для создания загрузчиков обучающих, проверочных и тестовых данных:

```
train_loader = create_dataloader(df_train)
val_loader = create_dataloader(df_val)
test_loader = create_dataloader(df_test)
```

14. На данный момент у нас есть три загрузчика данных, соответствующих обучающему, проверочному и тестовому наборам. Следующий шаг – реализация модели GNN.

Реализация гетерогенной GNN

В этом разделе мы реализуем гетерогенную GNN с помощью оператора GraphSAGE. Такая архитектура позволит нам учитывать оба типа узлов (хосты и потоки) для построения лучших эмбеддингов. Это достигается за счет дублирования и обмена сообщениями между различными слоями, как показано на рис. 17.5.

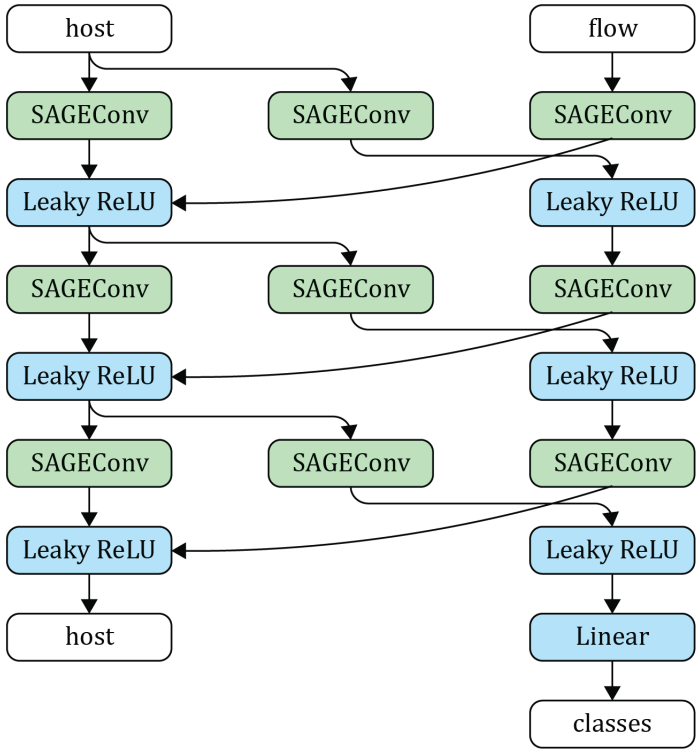


Рис. 17.5 ❖ Архитектура гетерогенной GNN

Мы реализуем три слоя SAGEConv с LeakyRELU для каждого типа узлов. Наконец, линейный слой выведет пятимерный вектор, в котором каждое измерение соответствует классу. Мы обучим эту модель, используя кросс-энтропию в качестве функции потерь и оптимизатор Adam.

1. Импортируем соответствующие слои нейронной сети из PyTorch Geometric:

```
from torch_geometric.nn import Linear, HeteroConv, SAGEConv, GATConv
```

2. Задаем гетерогенную GNN с тремя параметрами – количеством скрытых измерений, количеством выходных измерений и количеством слоев:

```
class HeteroGNN(torch.nn.Module):
    def __init__(self, dim_h, dim_out, num_layers):
        super().__init__()
```

3. Задаем гетерогенную версию оператора GraphSAGE для каждого слоя и типа ребра. Здесь мы можем применить для каждого типа ребра свой слой GNN, например GCNConv или GATConv. Обертка HeteroConv() управляет обменом сообщениями между слоями, как показано на рис. 17.5:

```
        self.convs = torch.nn.ModuleList()
        for _ in range(num_layers):
            conv = HeteroConv({
                ('host', 'to', 'flow'): SAGEConv((-1,-1), dim_h),
                ('flow', 'to', 'host'): SAGEConv((-1,-1), dim_h)
            }, aggr='sum')
            self.convs.append(conv)
```

4. Задаем линейный слой, который будет выводить итоговые результаты классификации:

```
        self.lin = Linear(dim_h, dim_out)
```

5. Задаем метод .forward(), который вычисляет эмбеддинги для узлов-хостов и узлов-потоков (хранятся в словаре x_dict). Затем эмбеддинги потоков используются для предсказания класса:

```
    def forward(self, x_dict, edge_index_dict):
        for conv in self.convs:
            x_dict = conv(x_dict, edge_index_dict)
            x_dict = {key: F.leaky_relu(x) for key, x in x_dict.items()}
        return self.lin(x_dict['flow'])
```

6. Создаем гетерогенную GNN с 64 скрытыми измерениями, 5 выходами (наши 5 классов) и 3 слоями. Если есть возможность, мы размещаем ее на GPU и задаем оптимизатор Adam с темпом обучения 0.001:

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = HeteroGNN(dim_h=64, dim_out=5, num_layers=3).to(device)
optimizer = Adam(model.parameters(), lr=0.001)
```

7. Задаем функцию `test()` и создаем массивы для хранения прогнозов и фактических меток. Мы также хотим подсчитать количество подграфов и общее значение функции потерь, поэтому создаем соответствующие переменные:

```
@torch.no_grad()
def test(loader):
    model.eval()
    y_pred = []
    y_true = []
    n_subgraphs = 0
    total_loss = 0
```

8. Мы получаем прогнозы модели для каждого батча и вычисляем значение функции потерь – кросс-энтропии:

```
    for batch in loader:
        batch.to(device)
        out = model(batch.x_dict, batch.edge_index_dict)
        loss = F.cross_entropy(out, batch['flow'].y.float())
```

9. Добавляем спрогнозированный класс в список прогнозов и делаем то же самое с фактическими метками:

```
        y_pred.append(out.argmax(dim=1))
        y_true.append(batch['flow'].y.argmax(dim=1))
```

10. Подсчитываем количество подграфов и общее значение функции потерь следующим образом:

```
        n_subgraphs += BATCH_SIZE
        total_loss += float(loss) * BATCH_SIZE
```

11. Теперь, когда итерирование по батчам завершено, вычисляем F1-меру (макро), используя списки прогнозов и фактических меток. Оценка F1-меры с макроусреднением является хорошей метрикой в условиях несбалансированного обучения, поскольку она одинаково учитывает все классы независимо от количества наблюдений:

```
    y_pred = torch.cat(y_pred).cpu()
    y_true = torch.cat(y_true).cpu()
    f1score = f1_score(y_true, y_pred, average='macro')
```

12. Мы возвращаем итоговое значение функции потерь, оценку F1-меры с макроусреднением, список прогнозов и список фактических меток:

```
    return total_loss / n_subgraphs, f1score, y_pred, y_true
```

13. Создаем цикл для обучения модели в течение 101 эпохи:

```
model.train()
for epoch in range(101):
    n_subgraphs = 0
    total_loss = 0
```

14. Мы обучаем гетерогенную GNN на каждом батче, используя кросс-энтропию в качестве функции потерь:

```
    for batch in train_loader:
        optimizer.zero_grad()
        batch.to(device)
        out = model(batch.x_dict, batch.edge_index_dict)
        loss = F.cross_entropy(out, batch['flow'].y.float())
        loss.backward()
        optimizer.step()

        n_subgraphs += BATCH_SIZE
        total_loss += float(loss) * BATCH_SIZE
```

15. Каждые 10 эпох мы оцениваем качество модели на проверочном наборе и выводим соответствующие метрики (функция потерь для обучающего набора, функция потерь для проверочного набора и оценка F1-меры с макроусреднением для проверочного набора):

```
    if epoch % 10 == 0:
        val_loss, f1score, _, _ = test(val_loader)
        print(
            f"Эпоха {epoch} \n| Функция потерь:"
            f"{total_loss/n_subgraphs:.4f} \n| "
            f"Функция потерь - проверочный набор: {val_loss:.4f} \n| "
            f"F1-мера - проверочный набор: {f1score:.4f}"
        )
```

```
Эпоха 0
| Функция потерь:0.1114
| Функция потерь - проверочный набор: 0.0068
| F1-мера - проверочный набор: 0.6598
Эпоха 10
| Функция потерь:0.0019
| Функция потерь - проверочный набор: 0.0019
| F1-мера - проверочный набор: 0.8809
Эпоха 20
| Функция потерь:0.0015
| Функция потерь - проверочный набор: 0.0016
| F1-мера - проверочный набор: 0.8970
Эпоха 30
| Функция потерь:0.0012
| Функция потерь - проверочный набор: 0.0013
| F1-мера - проверочный набор: 0.9287
Эпоха 40
| Функция потерь:0.0010
```

```
| Функция потерь - проверочный набор: 0.0012
| F1-мера - проверочный набор: 0.9320
Эпоха 50
| Функция потерь:0.0008
| Функция потерь - проверочный набор: 0.0010
| F1-мера - проверочный набор: 0.9545
Эпоха 60
| Функция потерь:0.0006
| Функция потерь - проверочный набор: 0.0009
| F1-мера - проверочный набор: 0.9570
Эпоха 70
| Функция потерь:0.0005
| Функция потерь - проверочный набор: 0.0010
| F1-мера - проверочный набор: 0.9668
Эпоха 80
| Функция потерь:0.0005
| Функция потерь - проверочный набор: 0.0009
| F1-мера - проверочный набор: 0.9728
Эпоха 90
| Функция потерь:0.0004
| Функция потерь - проверочный набор: 0.0010
| F1-мера - проверочный набор: 0.9723
Эпоха 100
| Функция потерь:0.0004
| Функция потерь - проверочный набор: 0.0010
| F1-мера - проверочный набор: 0.9720
```

16. Наконец, оцениваем качество модели на тестовом наборе. Мы также выводим отчет scikit-learn о классификации, который включает значение F1-меры с макроусреднением:

```
_, _, y_pred, y_true = test(test_loader)

print(classification_report(y_true, y_pred, target_names=labels, digits=4))
```

```
              precision    recall  f1-score   support

      benign     0.9999    0.9999    0.9999    700791
  bruteForce     0.9686    0.9506    0.9595       162
         dos     1.0000    1.0000    1.0000    125164
    pingScan     0.9050    0.9643    0.9337       336
    portScan     0.9941    0.9947    0.9944     18347

    accuracy                         0.9997    844800
   macro avg     0.9735    0.9819    0.9775    844800
weighted avg     0.9997    0.9997    0.9997    844800
```

Мы получили значение F1-меры с макроусреднением, равное 0.9735. Этот отличный результат показывает, что наша модель научилась надежно предсказывать каждый класс.

Примененный нами подход был бы еще более эффективным, если бы мы могли получить доступ к большему количеству признаков, связанных с хо-

стами, но достаточно уже того, что он показывает, как его можно расширить в соответствии с вашими потребностями. Еще одно главное преимущество GNN – способность обрабатывать большие объемы данных.

Такой подход имеет еще больший смысл при работе с миллионами потоков. Чтобы завершить этот проект, давайте построим график ошибок модели и посмотрим, как можно ее улучшить.

Мы создадим датафрейм данных для хранения прогнозов (`y_pred`) и фактических меток (`y_true`). Теперь воспользуемся им для построения круговой диаграммы, показывающей доли неправильно классифицированных наблюдений:

```
df_pred = pd.DataFrame([y_pred.numpy(), y_true.numpy()]).T
df_pred.columns = ['pred', 'true']
plt.pie(df_pred['true'][df_pred['pred'] != df_pred['true']].value_counts(),
        labels=labels, autopct='%.0f%%');
```

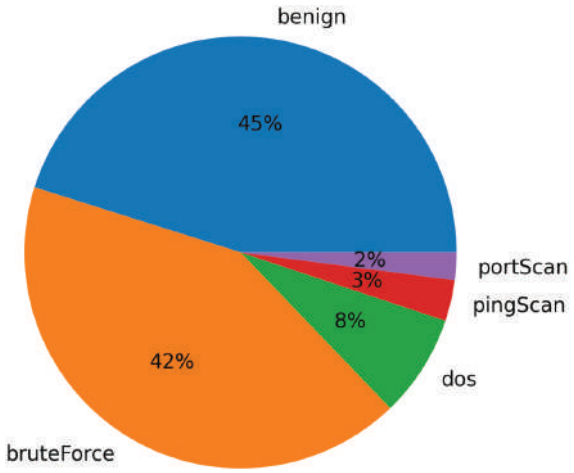


Рис. 17.6 ❖ Доля каждого неправильно спрогнозированного класса

Если мы сравним эту круговую диаграмму с исходными пропорциями в наборе данных, то увидим, что модель лучше справляется с мажоритарными классами. Это неудивительно, поскольку миноритарные классы труднее выучить (меньше наблюдений), а их пропуск штрафуется меньше (700 000 нормальных потоков против 336 пинг-сканирований). Обнаружение атак port scan и ping scan можно улучшить с помощью таких методов, как оверсемплинг и использование весов классов во время обучения.

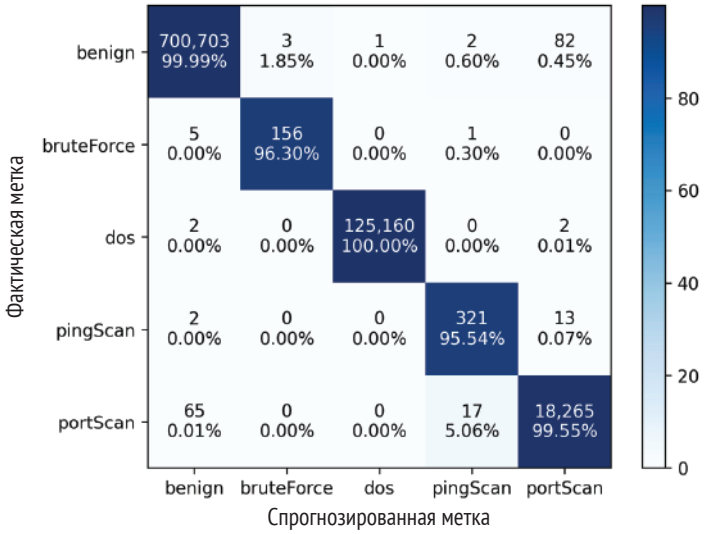
Мы можем получить еще больше информации, взглянув на матрицу ошибок.

```
matrix = confusion_matrix(y_true, y_pred)
norm_matrix = matrix / matrix.sum(axis=1) * 100

plt.imshow(norm_matrix, cmap='Blues')
```

```
plt.colorbar()
plt.xlabel('Спрогнозированная метка')
plt.ylabel('Фактическая метка')
plt.xticks(range(len(labels)), labels)
plt.yticks(range(len(labels)), labels)

for i, j in itertools.product(range(matrix.shape[0]), range(matrix.shape[1])):
    text = f"{matrix[i,j]:,}\n{norm_matrix[i,j]:.2f}%"
    plt.text(j, i, text,
            horizontalalignment='center', verticalalignment='center',
            color='white' if matrix[i,j] >= matrix[i,:].mean() else 'black')
plt.show()
```



Эта матрица ошибок показывает интересные результаты: смещение в сторону класса benign или ошибки прогнозирования атак ping scan как port scan и наоборот. Эти ошибки можно объяснить схожестью этих атак.

Конструирование дополнительных признаков могло бы помочь модели отличать эти классы друг от друга.

Выводы

В этой главе мы рассмотрели использование GNN для обнаружения аномалий в новом наборе данных – наборе данных `CIDDS-001`. Сначала мы провели предварительную обработку набора данных и преобразовали его в графовое представление, что позволило нам отразить сложные взаимосвязи между различными компонентами сети. Затем мы реализовали гетерогенную GNN с помощью операторов GraphSAGE. Это позволило учесть гетерогенность графа и классифицировать потоки как нормальные или вредоносные.

Применение GNN в области сетевой безопасности показало многообещающие результаты и открыло новые возможности для исследований. По мере развития технологий и увеличения объема сетевых данных GNN будут становиться все более важным инструментом для обнаружения и предотвращения нарушений сетевой безопасности.

Дополнительное чтение

[1] M. Ring, S. Wunderlich, D. Grüdl, D. Landes, and A. Hotho, *Flow-based benchmark data sets for intrusion detection, in Proceedings of the 16th European Conference on Cyber Warfare and Security* (ECCWS), ACPI, 2017, pp. 361–369.

Глава 18

Раскрытие потенциала графовых нейронных сетей в реальных задачах

Благодарим вас за то, что нашли время прочитать книгу «Руководство по графовым нейронным сетям в Python». Мы надеемся, что она дала вам ценные знания о мире графовых нейронных сетей и примерах их применения.

Завершая эту книгу, мы хотели бы дать вам несколько советов по эффективному использованию графовых нейронных сетей. При условии корректного использования GNN могут быть невероятно эффективными, но им свойственны те же преимущества и недостатки, что и другим методам глубокого обучения. Знать, когда и где применять эти модели, – важнейший навык, которым нужно овладеть, поскольку слишком сложные решения могут привести к низкой эффективности.

Во-первых, GNN особенно эффективны, когда для обучения доступен большой объем данных. Это связано с тем, что алгоритмы глубокого обучения требуют большого количества данных для эффективного изучения сложных закономерностей и взаимосвязей. При наличии достаточно большого набора данных GNN могут достичь высокого уровня точности и обобщения.

По тем же причинам GNN наиболее ценны при работе со сложными, высокоразмерными данными (характеристики узлов и ребер). Они могут автоматически исследовать сложные закономерности и взаимосвязи между признаками, которые человеку было бы трудно или невозможно выявить. Традиционные алгоритмы машинного обучения, такие как линейная регрессия или деревья решений, опираются на созданные вручную признаки, которые зачастую не в состоянии отразить всю сложность реальных данных.

Наконец, при работе с GNN важно убедиться, что представление графа добавляет ценность признакам. Это особенно актуально, когда граф представляет собой искусственно созданное, а не естественное представление, например социальные сети или белковые структуры. Связи между узлами не должны быть произвольными, а представлять собой значимые отношения между узлами.

Вы можете заметить, что некоторые примеры в этой книге не соответствуют ранее перечисленным правилам. В основном это связано с техническими ограничениями, связанными с возможностью запуска кода в Google Colab, а также с общим отсутствием высококачественных наборов данных. Однако они также отражают реальные наборы данных, которые могут быть беспорядочными и ограничены по объему. Большинство этих данных, как правило, можно представить в виде табличных наборов, на которых графовым нейронным сетям трудно превзойти ансамбли на основе деревьев типа XGBoost.

В целом правильные базовые решения имеют решающее значение, поскольку их сложно превзойти даже в подходящих условиях. Эффективная стратегия при работе с GNN заключается в реализации нескольких типов GNN и сравнении их качества. Например, GNN на основе сверток типа GCN (глава 6) могут хорошо работать с одними типами графов, в то время как GNN на основе механизма внимания типа GAT (глава 7) могут лучше подходить для других. Кроме того, в ряде задач хорошее качество может продемонстрировать GNN с передачей сообщений, например MPNN (глава 12). Обратите внимание, что каждый подход более выразителен, чем предыдущий, и у каждого есть свои сильные и слабые стороны.

Если вы работаете над более конкретной задачей, то в этой книге рассматривается несколько подходов GNN, которые могут оказаться более подходящими. Например, если вы имеете дело с небольшими данными о графах, в которых отсутствуют признаки узлов и ребер, вам стоит рассмотреть возможность использования Node2Vec (глава 4). Напротив, если вы имеете дело с большими графами, GraphSAGE и LightGCN помогут справиться с требованиями к времени вычислений и памяти (главы 8 и 17).

Кроме того, для задач классификации графов могут подойти GIN-слои и слои глобального пулинга (глава 9), а для прогнозирования связей можно использовать вариационные графовые автоэнкодеры и SEAL (глава 10). Для генерации новых графов можно изучить GraphRNN и MolGAN (глава 11). Если вы работаете с гетерогенными графами, то, возможно, вам стоит рассмотреть один из многочисленных вариантов гетерогенных GNN (главы 12 и 16). Для пространственно-временных графов могут быть полезны Graph WaveNet, STGraph и другие темпоральные GNN (главы 13 и 15). Наконец, если вам нужно объяснить прогнозы, сгенерированные вашей графовой нейронной сетью, можете обратиться к методам объяснения графов, рассмотренным в главе 14.

Прочитав эту книгу, вы получите глубокое представление о графовых нейронных сетях и о том, как их можно применять для решения реальных задач. По мере того как вы будете продолжать работать в этой области, мы призываем вас применять полученные знания на практике, экспериментировать с новыми подходами и продолжать наращивать свой опыт. Область машинного обучения постоянно развивается, и с течением времени ваши навыки будут становиться только ценнее. Мы надеемся, что вы будете применять полученные знания для решения проблем и оказания положительного влияния на мир. Еще раз благодарим вас за прочтение этой книги и желаем всего наилучшего в ваших будущих начинаниях.

Предметный указатель

Д

И

К

Л

М

Н

Максим Лабонн, Артем Груздев

Графовые нейронные сети на Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Груздев А. В.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 27,79. Тираж 100 экз.

Веб-сайт издательства: **www.dmkpress.com**

Графовые нейронные сети стали одной из самых интересных архитектур в глубоком обучении. Технологические компании теперь пытаются применить их повсюду: в системах рекомендаций еды, видео и поиска романтических партнеров, для выявления фейковых новостей, проектирования микросхем и 3D-реконструкции.

В процессе чтения вы научитесь:

- создавать графовые наборы данных из табличных или исходных данных;
- преобразовывать узлы и ребра в высококачественные эмбеддинги;
- реализовывать графовые нейронные сети с использованием PyTorch Geometric;
- выбирать лучшую модель графовых нейронных сетей в зависимости от вашей задачи;
- выполнять такие задачи, как классификация узлов, генерация графов, предсказание связей.

Издание предназначено специалистам по анализу и обработке данных, а также будет полезно разработчикам на Python и студентам вузов, желающих приобрести знания по одной из самых популярных архитектур ИИ. Для изучения материала пригодятся базовые знания языка Python и линейной алгебры.

Максим Лабонн – старший научный сотрудник в области машинного обучения в Liquid AI, возглавляющий посттренинговую подготовку. Признан экспертом Google Developers в области ИИ/МО. Активный блогер, внес значительный вклад в развитие ряда инструментов и современных моделей, в частности LLM AutoEval, NeuralBeagle и Phixtral.

Артем Груздев – основатель и директор компании «Гевисста», имеет 10-летний опыт прогнозирования кредитных рисков и 18-летний опыт статистического анализа. В последние годы активно занимается практическим построением прогнозных моделей. Ведет Telegram-канал, посвященный машинному обучению: https://t.me/Gewissta.

ИЦ «Гевисста» Исследовательский центр «Гевисста» с 2009 г. осуществляет разработку, валидацию, внедрение и мониторинг риск-моделей, моделей оттока, моделей отклика, в том числе на R и Python. Осуществляет подготовку специалистов в сфере прогнозного моделирования и анализа данных.

Packt›

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

www.дмк.рф

ISBN 978-5-93700-319-5

9 785937 003195 >