Федеральное государственное образовательное бюджетное учреждение высшего образования

«ФИНАНСОВЫЙ УНИВЕРСИТЕТПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»

(Финансовый университет)

Департамент анализа данных, принятия решений и финансовых технологий

Курсовая работа

на тему

«Классификация узлов сети с помощью графовой нейронной сети, использующей атрибуты узлов»

Выполнил:

студент группы ПМ19-1

Волненко А.А.

Научный руководитель:.

Быков А.А

Москва 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В последнее время все больше и больше внимания уделяется изучению графов. Такой растущий интерес к данной структуре данных связан с тем, что с помощью графа можно представить новые виды данных, такие как социальные сети, структура молекулы, географическая карта и другие. Помимо данных примеров, которые обладают естественной структурой схожей с графовой, текст и изображения также могут быть представлены в подобном виде. Важность и уникальность графовой структуры данных заключается в том, что она позволяет работать с такими абстрактными понятиями как отношения и взаимодействия отдельных объектов. С растущей вычислительной мощностью компьютеров в последние годы также стало популярным машинное обучение на графах, а в частности графовые нейронные сети (GNNs).

Большинство практических задач, которые решаются с помощью графовых нейронных сетей можно разделить на три типа: классификация вершин, предсказания ребер и классификация самого графа. Примером классификации вершин графа может быть предсказание является ли участник социального графа ботом или реальным человеком. Построение рекомендательной системы является задачей предсказания возможных ребер в графе, а примером классификации графа может быть предсказание свойств молекулы в зависимости от ее структуры.

Данная работа рассматривает задачу классификации узлов в графе с помощью атрибутов сети. Поставленная проблема является актуальной, потому что в последнее время графовые нейронные сети находят все большее применение в практических сферах деятельности.

Глава 1. Теоретическая справка

1.1 Базовые понятия

1.1.1. Что такое граф

Граф - распространенная структура данных, которая может быть использована для описания сложных система. В более общем понимании, граф это набор объектов (вершин, узлов), а также множество связей (ребер) между парами этих объектов. Например, чтобы представить социальную сеть в качестве графа, мы можем использовать людей в качестве вершин, а для того, чтобы показать, что два человека являются друзьями, будем добавлять ребро между ними.

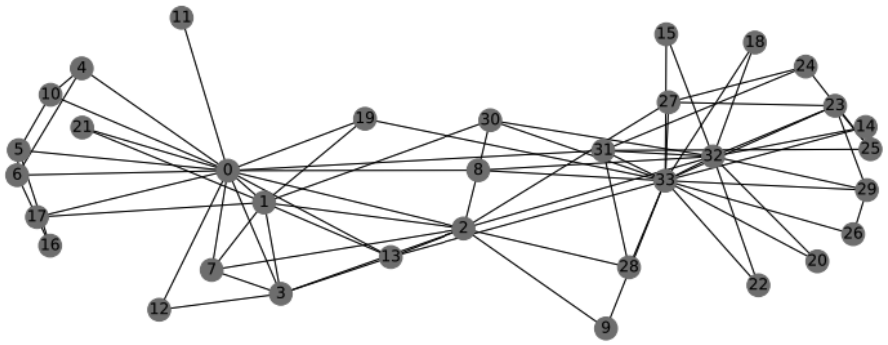


Рисунок 1: Одна из возможных визуализаций популярного датасета Zachary Karate Club Network

Так, Преимущество использования графовых структур данных состоит в том, что они учитывают как свойства отдельных объектов, так и отношения между ними. С помощью графа можно решать множество задач: представление социальной сети, взаимодействие белков и медицинских препаратов, отношения атомов в молекуле и так далее.

Формально граф $G = (V, E)$ определяется как множество вершин V и ребер между ними E. Ребро, соединяющее одну вершину $u \in V$ с другой вершиной $v \in V$ обозначается как $(u, v) \in E$. Удобным способом

представления графа является матрица смежности $A \in R^{|V| \times |V|}$. Для представления графа с помощью матрицы смежности необходимо упорядочить вершины так, чтобы номер каждой вершины соответствовал номеру строки и колонки в данной матрице. Наличие связи между двумя вершинами устанавливается через записи в матрице смежности: $A[u, v] = 1$ если $(u, v) \in E$ и $A[u, v] = 0$ в ином случае. Если граф содержит только ненаправленные ребра, то матрица смежности будет симметричной. В некоторых графах каждая связь может иметь некоторый вес, тогда матрица смежности будет содержать произвольные вещественные значения, а не только из множества $\{0, 1\}$. Например, в графе, который рассматривает взаимодействие протеинов, вес ребра может указывать на силу связи двух протеинов

Помимо различий в направленности ребер и наличии у них весов графы так же могут различаться по наличию у них связей между вершинами различного типа. Так, если рассматривать граф, представляющий взаимодействие двух препаратов, тогда различные виды связей будут соответствовать различным побочным эффектам, когда эти два препарата принимаются одновременно. Для того, чтобы учесть такие случаи добавим τ для обозначения типа связи, тогда каждое ребро графа будет представляться в следующем виде: $(u, \tau, v) \in E$, тогда для каждого вида связи будет определена своя матрица смежности A_τ. Графы такого типа называются графами с различными видами связями (multi-relational graphs) и весь граф может быть представлен как тензор смежности: $A \in R^{|V| \times |R| \times |V|}$, где $|R|$ - кол-во типов связи.

1.1.2. Задача классификации узлов

Задачей классификации узлов является предсказание метки объекта - y_u, которая может быть типом, категорией или атрибутом объекта. Задача классификации узлов является одной из самых популярных задач в машинном

обучении на графах, особенно в последние года. Примерами решения данной задачи на практике может быть нахождение ботов в графе социальной сети, классификация темы документа на основе основе графа цитирования, а также классификация белков в итерактоме.

На первый взгляд может показаться, что классификация узлов является обычным примером задачи обучения с учителем, однако тут не все так просто и между ними есть существенных различия. Главное отличие заключается в том, что узлы в графе не являются независимыми одинаково распределенными (i.i.d.). Обычно, при построении модели машинного обучения в задаче обучения с учителем делается предположение, что все объекты являются независимыми друг от друга, ведь иначе приходилось бы также моделировать зависимости между каждой парой объектов. Так же делается предположение о том, что все объекты одинаково распределены, ведь иначе нельзя было бы гарантировать, что построенная модель будет применима к новым объектам. В задаче классификации узлов нарушаются все вышеперечисленные предположения, потому что моделируются не одинаково распределенные объекты, а множество взаимосвязанных узлов.

Большинство успешных методов классификации узлов в графе непосредственно используют связи между вершинами. Одной из популярных идей является использование гомофилии, т.е. вершины, которые находятся близко друг к другу обладают схожими атрибутами. Например, люди, которые состоят в дружественных отношения, склонны разделять интересы друг друга. Основываясь на данном предположении можно строить модели, которые будут назначать метки, схожие с теми, которыми обладает окружение вершины.

1.2. Распространение сообщений (Message Passing)

Основная проблема, возникающая при обработке графовых данных состоит в том, что для них невозможно использовать уже существующие архитектуры нейронных сетей, так сверточные нейронные сети (CNNs)

применимы только к сеточно структурированным данным, таким как изображения, а рекуррентные нейронные сети (RNNs) определены только для последовательных входных данных, таких как текст. Поэтому, необходимо определить новую архитектуру нейронных сетей для работы с графовыми структурами данных.

Для начала рассмотрим механизм работы распространения сообщений (message passing), потому что он лежит в основе любой графовой нейронной сети, независимо от ее вариации. Итак, на каждой итерации распространения сообщений эмбеддинг (hidden embedding) $h_u^{(k)}$, соответствующий узлу $u \in V$, обновляется, используя информацию, которая содержится в соседях данной вершины $N(u)$. Формула для обновления эмбеддинга очередной вершины графа может быть записана следующим образом:

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \\ &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right),\end{aligned}$$

где UPDATE и AGGREGATE - произвольные дифференцируемые функции, а $m_{N(u)}$ - “сообщение”, которое является результатом агрегации информации от соседей вершины u. Надстрочные индексы используется для указания номера итерации распространения сообщений.

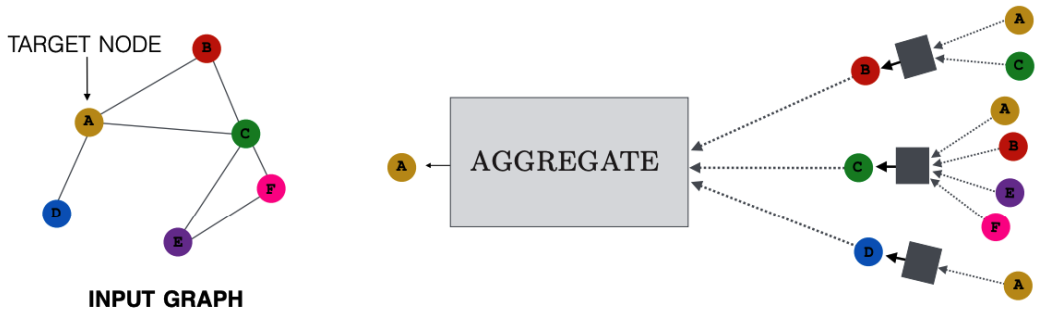
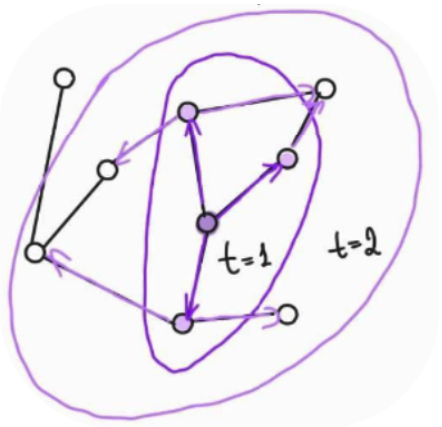


Рисунок 2. Иллюстрация того, как одна вершина собирает информацию от своих соседей. В данном случае отображается результат работы двухслойного распространения ошибки.

На каждой итерации k функция AGGREGATE принимает в качестве входных данных множество эмбеддингов $N(u)$, которое является соседями вершины u , а на выходе получает значение сообщения $m^{(k)}_{N(u)}$, на основе собранной информации от соседей. Так как функция AGGREGATE принимает на вход множество значений, то она по определение нечувствительна к различным перестановкам входных значений. Функции UPDATE объединяет значение $m^{(k)}_{N(u)}$ с эмбеддингом предыдущий итерации $h_u^{(k-1)}$ вершины u, чтобы получить обновленный эмбеддинг $h_u^{(k)}$. Изначальное значение эмбеддинга равно входному значению признака вершины, т.е. $h_u^{(0)} = x_u, \forall u \in V$.

По сути, смысл распространения сообщений в том, что на каждой итерации каждая вершина собирает информации от своих соседей, и по мере того, как номер итерации увеличивается, эмбеддинг каждой вершины будет содержать информации от все более дальних узлов. Так, после первой итерации ($k = 1$), каждая вершина содержит информацию от узлов, которые находятся на удаленности в одно ребро, после второй итерации ($k = 2$) каждая вершина будет уже содержать информацию от всех узлов, которые находятся на расстоянии в два ребра. В общем случае получаем, что после k-й итерации эмбеддинг каждой вершины содержит информацию о всех ее соседях на удаленности в k ребер.



Изображение 3. Иллюстрация итераций метода распространения ошибки

До этого момента приводились абстрактная модель графовой нейронной сети, основанная на модели распространения сообщений, использовав функции UPDATE и AGGREGATE. Для того, чтобы перевести эту модель во что-то, что можно было бы применить на практике достаточно просто подставить конкретные функции вместо абстрактных UPDATE и AGGREGATE. Классические графовые нейронную сети обновляют состояние эмбеддинга по следующей формуле:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\sum_{v\in\mathcal{N}(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right),$$

где $W_{self}^{(k)}$, $W_{neigh}^{(k)} \in R^{d^{(k)} * d^{(k-1)}}$ - обучаемые параметры сети, а σ обозначает поэлементную нелинейную функцию, такую как например tahn или ReLU. Значение баеса $b^{(k)} \in R^{d^{(k)}}$ обычно опускается для простоты записи, однако его добавление в формулу важно для хорошей работы сети.

Мы также можем эквивалентно определить данную классическую графовую нейронную сеть через функции UPDATE и AGGREGAT:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v\in\mathcal{N}(u)}\mathbf{h}_v,$$

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)}\right),$$

где “сообщение” может быть представлено в виде функции AGGREGATE следующим образом:

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

В качестве упрощенного варианта модели распространения сообщений можно рассматривать подход, в котором мы добавляем для каждой вершины

ребра, которые ведут в саму же эту вершину, таким образом мы сможем избавиться от непосредственного использования функции UPDATE. Таким образом, можно обновление эмбеддинга можно записать в следующем виде:

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

где теперь агрегация происходит над множеством $N(u) \cup \{u\}$, т.е. теперь мы агрегируем информацию от непосредственной вершины, так и ее соседей. Преимущество такого подхода в том, что теперь нет необходимости отдельно вводить функцию UPDATE, потому что она неявным образом используется во время агрегации. Упрощая таким образом модель распространения сообщений, мы уменьшаем шанс переобучить нашу модель, однако одновременно и ограничиваем ее возомжности, потому что теперь мы не различаем, приходит ли информация от самой вершины или ее соседей.

1.3. Обобщенная функция агрегации соседей

Классическая графовая нейронная сеть может показывать хорошие результаты, однако как и классическая MLP или RNN, она также может быть улучшена. Далее рассмотрим как это может быть сделано через обобщение и изменение функций AGGREGATE.

1.3.1. Нормализация соседних узлов

Самая базовая функция агрегации просто берет сумму эмбеддингов сосдених узлов. Однако этот подход может привести к численным проблемам при оптимизации, потому что количество соседей может сильно различаться, от чего значение суммы будет сильно колебаться.

Одно из решений данной проблемы - это просто отнормировать функцию агрегации по числу соседей, т.е. теперь мы будет брать не сумму значений, а их среднее:

$$\mathbf{m}_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v}{|\mathcal{N}(u)|},$$

однако и другие варианты нормирования показывают хорошие результаты, один из таких примеров, это симметричная нормализация:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}.$$

1.3.2. Графовая сверточная сеть (GCN)

Одной из наиболее популярных вариаций графовых нейронных сетей является графовая сверточная сеть. Данный подход использует симметричную нормализацию в функции агрегации, а также подход с петлями у вершин. Такая сеть определяет модель распространения сообщений следующим образом:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right).$$

1.3.3. Агрегация на множестве

Нормализация информации от соседних вершин дает хорошее прирост к качеству модели, однако можно ли как-то еще улучшить функцию AGGREGATE, есть ли что-то более сложное, чем обычная сумму соседей? Сама по себе данная функция принимает на вход множиство эмбеддингов соседей вершины h_v, $\forall v \in N(u)$ и должна преобразовывать его в один вектор $m_{N(u)}$. Факт того, что данная функция принимает на вход именно множество - важен, потому что это означает, что любая агрегирующая функция должна быть инвариантна к перестановкам своих аргументов.

1.3.3.1 Set pooling

Один из подходов к определению функции агрегации называется set pooling, который использует многослойные персептроны (MLPs):

$$\mathbf{m}_{\mathcal{N}(u)} = \mathrm{MLP}_\theta \left(\sum_{v \in N(u)} \mathrm{MLP}_\phi(\mathbf{h}_v) \right),$$

Так же было доказано, что данная функция агрегации является универсальным аппроксиматором, т.е. это означает, что любая функция агрегации, инвариантная к перестановкам своих аргументов, которая принимает на вход множество эмбедингов и возвращает один вектор, может быть приближена с желаемой точностью с помощью модели, использующей данную формулу.

В изначальной статье использовалась сумма эмбеддингов после того, как к ним применили многослойный персептрон, однако ее можно заменить и иной функций, такой как поэлементный максимум или минимум.

Данный подход приводит к небольшому улучшению качества модели, однако и увеличивает вероятность переобучения в зависимости от глубины персептронов, которые используется, поэтому обычно берут только однослойные сети, т.к. их достаточно, чтобы удовлетворить требованиям моделя и одновременно не сильно увеличивать риск переобучения.

1.3.3.2. Janossy pooling

Предыдущий подход по сути просто добавлял многослойные персептроны поверх базовых функций агрегации, однако существует совершенно иной способ агрегирования информации, нежели чем простое использование суммы или среднего эмбедингов соседей вершины, который называется Janossy pooling.

Для начала вспомним, что мы должны использовать инвариантные к перестановкам своих аргументов функции, потому что среди соседей вершины изначально не подразумевается никакого порядка. Однако Janossy pooling использует совершенно иной подход: вместо того, чтобы использовать

инвариантные к перестановкам функции, будут применяются функции, чувствительные к порядку своих аргументов, а потом результаты всех возможных перестановок будут усреднены

Пусть $\pi_i \in \Pi$ обозначает функцию перестановки, которая преобразует множество $\{h_v, \forall v \in N(u)\}$ в определенную последовательность $(h_{v1}, h_{v2}, \ldots, h_{v|N(u)|})_{\pi_i}$. Другими словами π_i принимает на вход неупорядоченную последовательность эмбеддинг соседей вершины, а на выходе возвращает эти же эмбеддинги в определенном порядке. Janossy pooling определяет функцию агрегации соседей следующим образом:

$$\mathbf{m}_{\mathcal{N}(u)} = \mathrm{MLP}_\theta \left(\frac{1}{|\Pi|} \sum_{\pi \in \Pi} \rho_\phi \left(\mathbf{h}_{v_1}, \mathbf{h}_{v_2}, \ldots, \mathbf{h}_{v_{|\mathcal{N}(u)|}} \right)_{\pi_i} \right),$$

где Π обозначает множество перестановок, а p_ϕ - это функция, чувствительная к порядку своих аргументов. На практике в качестве данной функции используются модели LSTM.

В изначальной формуле подразумевается, что Π - это множество всех возможных перестановок, тогда и функция агрегации - универсальный аппроксиматор функций над множествами, как в подходе set pooling. Однако, суммирование по всем возможным вариантам перестановки - достаточно долгая и трудоемкая операция, поэтому на практике обычно используют следующий подход: выбирается случайное подмножество перестановок во время каждого применения функции агрегации и производится суммирование только по этому подмножеству.

В ряде синтетических тестов было показано, что Janossy pooling дает небольшой прирост в качестве относительно set pooling.

1.3.4 Модель внимания для графовой нейронной сети

Вдобавок к предыдущим двум методом достаточно популярным подходом для улучшения функции агрегации является использование в ней модели

внимания. По сути, мы назначаем каждой вершине ее вес, который будет использоваться на шаге агрегации. Тогда функция “сообщения” будет выглядеть следующим образом:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v,$$

где $a_{u,v}$ является весом внимания для соседа $v \in N(u)$, когда мы агрегируем информацию для вершины u. В изначальной статье веса внимания определены следующим образом:

$$\alpha_{u,v} = \frac{\exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v]\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}]\right)},$$

где a - обучаемый вектор внимания, W - обучаемая матрица, а $\oplus$ обозначает операцию конкатенации.

Существуют также и другие варианты вычисления весов внимания, которые обычно используются в других архитектурах нейронных сетей, но могут быть применены и в нашей случае:

$$\alpha_{u,v} = \frac{\exp\left(\mathbf{h}_u^\top \mathbf{W}\mathbf{h}_v\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{h}_u^\top \mathbf{W}\mathbf{h}_{v'}\right)},$$

либо также есть вариант, использующий многослойные персептроны:

$$\alpha_{u,v} = \frac{\exp\left(\mathrm{MLP}(\mathbf{h}_u, \mathbf{h}_v)\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathrm{MLP}(\mathbf{h}_u, \mathbf{h}_{v'})\right)},$$

где MLP имеют одно выходное значение.

В общем случае, добавление модели внимание в графовую нейронную сеть повышает возможности сети, особенно в тех случаях, когда мы знаем, что определенные вершины обладают большей информативностью, нежели другие.

1.4. Обобщенная функция обновления состояния

Функция AGGREGATE получила большое внимание со стороны исследователей, потому что именно через ее вариации были предложены новые разновидности графовых сетей. Наиболее ясно это стало, после того, как была представлена одна из наиболее популярный графовых нейронных сетей GraphSage, которая использовала обобщение для функции агрегации соседей. Однако механизм распространения сообщения содержит в себе два шага: агрегация информации от соседей и обновление состояния вершины и во многих случаях функция UPDATE также играет важную роль в определении качества модели сети.

Прежде чем перейти к описанию обобщенных функций обновления состояния стоит упомянуть о важной проблеме, которая может возникнуть во время обучения сети - чрезмерном сглаживании (over-smoothing). Идея состоит в том, что после нескольких итераций распространения сообщений вершины графа могут стать сильно похожими друг на друга. Такой эффект чаще всего наблюдается в самых простых графовых нейронных сетях и тех подходах, которые используют петли для обновления состояния вершин. Проблема чрезмерного сглаживания важна, потому что она не позволяет строить многослойные сети, которые необходимы для отслеживания связей вершин, находящихся на удаленном расстоянии.

1.4.1. Метод пропуска связи (skip-connections)

Как было сказано ранее проблема чрезмерного сглаживания - ключевая проблема графовых нейронных сетей. Она случается, когда информация присущая конкретной вершине оказывается потеряна после нескольких итераций распространения сообщений. По сути, мы можем ожидать появление данной проблемы, когда информация от соседей начинает преобладать в обновлении состояния вершины. Для решения решения этой проблемы мы можем использовать метод пропуска связи, который непосредственно пытается сохранить состояние вершины из предыдущих итераций распространения

сообщений. Данный подход может использоваться совместно с иными методами обновления состояния вершины и чаще всего идет просто как дополнение или надстройка над другими, поэтому будем использовать $UPDATE_{base}$ для обозначения исходной функции обновления, поверх которой и будет применять метод пропуска связи. Тогда в простейшем варианте этого метода можем использовать обычную конкатенацию векторов для функцию обновления состояния вершины:

$$\mathrm{UPDATE}_{\mathrm{concat}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = [\mathrm{UPDATE}_{\mathrm{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) \oplus \mathbf{h}_u],$$

где мы по сути просто конкатенируем результат изначальной функции обновления со значением вершины на предыдущей итерации.

Использование конкатенации векторов было впервые предложено в модели GraphSage, которая одна из первых показала преимущества применения модификаций функции обновления состояния. Однако помимо конкатенации векторов мы можем использовать и другие вариации метода пропуска связи, например линейную интерполяцию:

$$\mathrm{UPDATE}_{\mathrm{interpolate}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \boldsymbol{\alpha}_1 \odot \mathrm{UPDATE}_{\mathrm{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) + \boldsymbol{\alpha}_2 \odot \mathbf{h}_u,$$

где $a_1, a_2 \in [0, 1]^d$ и $a_2 = 1 - a_1$, а $\odot$ означает поэлементное умножение. В данном подходе конечное обновленное состояние является линейной интерполяцией между предыдущим значением вершины и обновленным значением, основанным на информации от соседей. Параметр a_1 обучается вместе с остальными параметрами модели.

1.4.2. Gated Updates

Одним из принципиально отличных подходов к обновлению состояние вершины было вдохновлено рекуррентными нейронными сетями. В этом случае один из способов представления механизма распространения сообщения в графовых нейронных сетях заключается в том, что на очередной его итерация

функция обновления получает агрегированную информацию от соседей в качестве нового наблюдения, которое в дальнейшем используется для вычисления следующего нового состояния вершины. Тогда мы можем непосредственно заимствовать методы, которые используются в рекуррентных сетях для обновления состояния. Так в одном из первых вариантов графовой нейронной сети функция UPDATE была определена следующим образом:

$$\mathbf{h}_u^{(k)} = \mathrm{GRU}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}),$$

где GRU обозначает механизм обновления состояния в вентильных рекуррентных нейронных сетях (GRU). Также существуют подходы, использующие функции обновления состояния основанные на архитектуре LSTM.

В общем случае, любая функция обновления состояния, которая используется в рекуррентных нейронных сетях может быть использована и в графовых тоже. Мы просто заменяем в функции обновления текущее состояние в рекуррентной сети на текущее состояние вершины в графе, а вектор наблюдения заменяется на вектор “сообщения” от соседей.

Данный подход эффективен для построения глубоких графовых нейронных сетей (т.е. содержащие 10 и более слоев) и предотвращения чрезмерного сглаживания.

1.4.3. Jumping Knowledge Connection

До этого момента подразумевалось, что конечное значение вершины было равно ее состоянию после последней итерации метода распространения сообщений, то есть:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}.$$

Данное предположение свойственно многим вариациям графовых сетей, однако недостаток такого подхода в том, что он приводит к проблеме чрезмерного сглаживания.

Тогда для того, чтобы улучшить качество конечного значения вершины мы можем использовать альтернативный подход, который заключается в том, чтобы использовать представления вершины на каждом слою метода распространения сообщений. Тогда конечное значение вершины может определяться следующим образом:

$$\mathbf{z}_u = f_{\text{JK}}(\mathbf{h}_u^{(0)} \oplus \mathbf{h}_u^{(1)} \oplus ... \oplus \mathbf{h}_u^{(K)}),$$

где f_{JK} - это произвольная дифференцируемая функция. Такой подход известен как добавление jumping knowledge (JK) connections и был впервые предложен в 2018 году. Во многих подходах в качестве функции f_{JK} используется простое тождественное отображение, т.е. мы просто используем конкатенацию эмбеддингов вершины с каждого слоя, однако и другие подходы могут быть использованы, например, map-pooling или LSTM. В целом, данный подход приводит к стабильному улучшению качества в ряде задач и поэтому является полезным для применения на практике.

1.5. Нейронные сети с разными типами связей

До этого момента были рассмотрены нейронные сети, которые подразумевали, что мы используем простые графы. Однако встречаются задачи, в которых графы могут обладать разными типами связей, например, графы знаний. Далее рассмотрим наиболее популярный подход, который применяется для подобных задач.

1.5.1. Реляционная графовая сверточная сеть (RGCN)

Данный подход был предложен в 2017 году и известен как реляционная графовая сверточная сеть (Relational Graph Convolutional Network или просто RGCN). В данном случае предлагается изменить функцию агрегации информации от соседей, добавляя отдельную матрицу преобразований для каждого типа связи:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{\tau \in \mathcal{R}} \sum_{v \in \mathcal{N}_\tau(u)} \frac{\mathbf{W}_\tau \mathbf{h}_v}{f_n(\mathcal{N}(u), \mathcal{N}(v))},$$

где f_n - функция нормализаци, которая может зависеть как от соседей вершины u, так и соседей вершины v, которая сама является непосредственным соседом вершины u. По сути, данный подход является аналогом классической графовой нейронной сети использующей нормализацию, однако в этом случае мы отдельно собираем информацию от соседей с разными типами связей.

1.6. Обучение сети и функция потерь

Классический подход для применения графовых нейронных сетей в задаче классификации узлов заключается в том, что производится обучение с учителем, где функционал ошибки определяется, используя softmax и логарифмическую функцию потерь:

$$\mathcal{L} = \sum_{u \in \mathcal{V}_{\text{train}}} -\log(\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)).$$

Здесь предполагается, что $y_u \in Z^c$ - это one-hot вектор, указывающий на значение истинного класса узла из обучающей выборки $u \in V_{train}$. Для того, чтобы вычислить вероятность принадлежности узла к классу к его классу - y_u, была использована функция $softmax(z_u, y_u)$, которая определяется следующим образом:

$$\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) = \sum_{i=1}^{c} \mathbf{y}_u[i] \frac{e^{\mathbf{z}_u^\top \mathbf{w}_i}}{\sum_{j=1}^{c} e^{\mathbf{z}_u^\top \mathbf{w}_j}},$$

где $w_i \in R^d$, $i = 1, \ldots, c$ - обучаемые параметры сети. Существую и другие вариации функционалов потерь для задачи обучения с учителем для

классификации узлов, однако приведенный выше подход используется наиболее часто.

1.7. Регуляризация

Применение методов регуляризации крайне важно для всех нейронных сетей и графовые архитектуры не являются исключением. Большинство известных методов регуляризации подходят для работы с графовыми нейронными сетям, однако существуют и другие подходы, присущие только данному виду архитектур. Далее будут рассмотрены наиболее часто применимы методы регуляризаций для графовых сетей.

1.7.1. Edge Dropout

В данном подходе регуляризации мы случайным образом удаляем (маскируем) ребра графа в матрице смежности во время процесса обучения с целью сделать нашу нейронную сеть менее склонной к переобучению и более устойчивой к шуму в матрице смежности. Данный подход наиболее часто применяется для графов знаний, а так же был использован в изначальной версии графовой нейронной сети с механизмом внимания.

Глава 2. Практическая часть

Для практической части был выбрал Cora датасет, на котором была произведена классификация узлов, с помощью графовой нейронной сети. Данный датасет состоит из 2708 научных публикаций, которые являются узлами в графе. Каждая публикация относится к одному из семи классов, которые являются темами публикаций, например, “нейронные сети”, “вероятностные методы”, “обучение с подкреплением” и другие. Пара вершин обладает ребром между собой, если одна публикация ссылается на другую. Все ребра в графе являются ненаправленными, т.е. неизвестно какая именно статья из двух ссылается на другую.

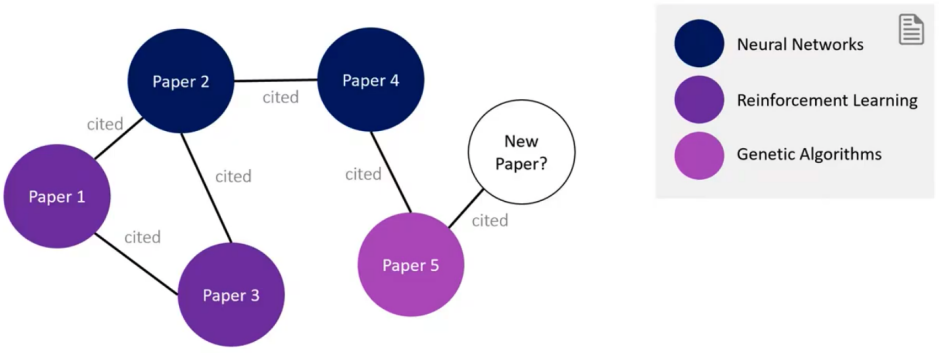


Рисунок 4. Иллюстрация небольшой части датасета Cora и задачи классификации узлов.

Каждая вершина графа обладает набором признаков, который описывает содержание соответствующей научной публикации. Каждый набор признаков - это мешок слов (bag of words) размерности 1433. Т.е. для описания научных статей были выбраны 1433 определенных слов. Далее для характеризации конкретной статьи производится подсчет количества каждого из выбранных 1433 слов, например слово Network встречается 7 раз, слово ReLU 1 раз и тд. Далее все полученные значения в мешке слов нормализуются.

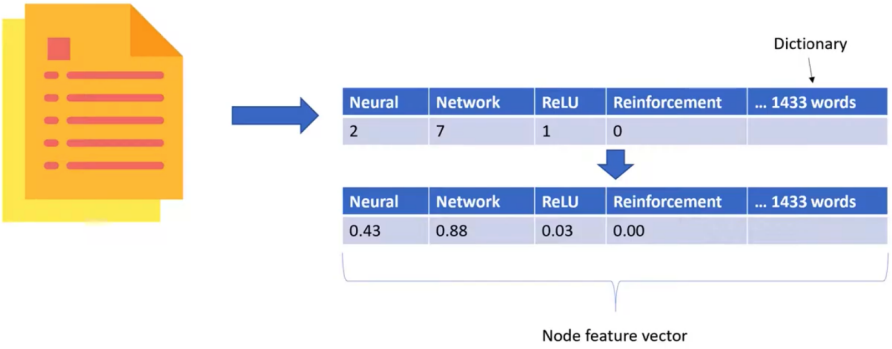


Рисунок 5. Иллюстрация набора признаков - мешка слов для одной вершины.

Также стоит отметить, что для подсчета ошибки на обучающей, валидационной и тестовой выборке используются специальные бинарные маски, которые используются для отбора соответствующих объектов. Бинарная маска - вектор, на i-й позиции которого стоит 1, если i-й объект принадлежит соответствующей выборке и 0 иначе.

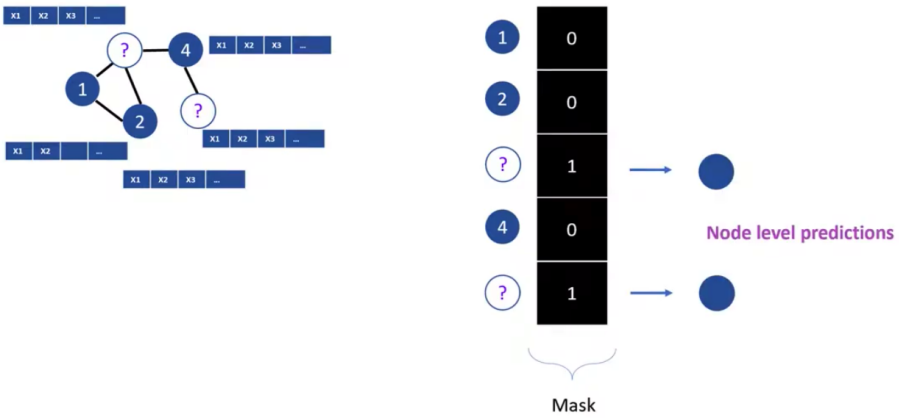


Рисунок 6. Иллюстрация возможного представления бинарной маски для тестовой выборки.

Теперь перейдем к определению графовой нейронной сети, для построения которой использовалась библиотека PyTorch Geometric. Для данной задачи была использована нейронная сеть, описанная в статье Kipf and Welling, 2016, которая содержит два слоя, на каждом из которых используются графовые сверточные слое (GCN's). Помимо этого для регуляризации на каждый слой был добавлен dropout, который используется только во время обучения. Значения

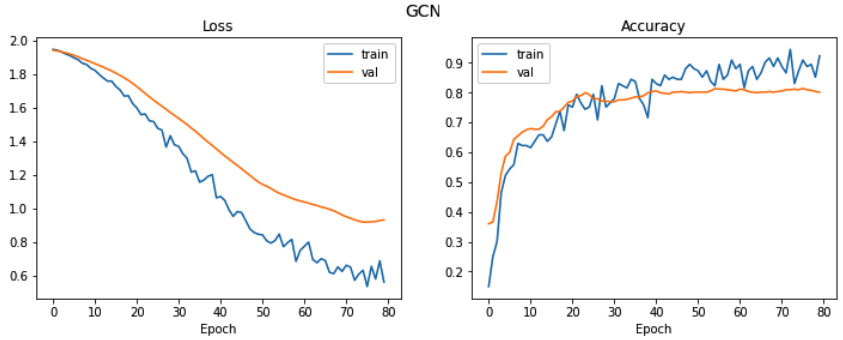
гиперпараметров были взяты из той же статье, только кол-во нейронов на скрытом слое было изменено с 16 на 64, потому что это привело к небольшому улучшению качества классификации. Тогда построенная нейронная сеть выглядит следующим образом:

```
Graph Convolutional Network (GCN):
GCN(
  (dropout1): Dropout(p=0.5, inplace=False)
  (conv1): GCNConv(1433, 64)
  (relu): ReLU(inplace=True)
  (dropout2): Dropout(p=0.5, inplace=False)
  (conv2): GCNConv(64, 7)
)
```

В качестве функции потери использовалась кросс-энтропийная функция потерь. Максимальное число итераций обучения равно 200, само же обучение производилось с помощью оптимизатора Adam с коэффициентом скорости обучения равным 0.01. Также использовалась операция преждевременной остановки (early stopping), которая останавливает процесс обучения модели, если ошибка на валидационной выборке за последние 10 итераций не уменьшается.

Процесс обучения завершился на 79 итерации после преждевременной остановки. Точность на контрольной выборке составила 81%. Более подробная информация представлена ниже:

```
Epoch: 79
----------
Train loss: 0.5608 | Train acc: 0.9214
  Val loss: 0.9312 |   Val acc: 0.8000
 Test loss: 0.8899 |  Test acc: 0.8160
```



Достигнутая точность является хорошим результатом и схожа с той, что была получила в статье Kipf and Welling, 2016. На данный момент лучшая точность классификации на датасете Cora является 90%.

ЗАКЛЮЧЕНИЕ

Целью данной курсовой работы было рассмотрение задачи классификации узлов графа с помощью нейронных сетей. В практической части работы была выполнена классификация узлов графа из датасета Cora с помощью двухслойной сверточной графовой сети, а точность классификации составила 81%. В целом графовые нейронные сети набирают популярность в последнее время, поскольку появляются все новые виды задач, которые можно решить с помощью графовых структуры данных. Безусловно данный вид архитектур в ближайшем будущем будет только все больше развиваться и находить новые приложения в практической сфере деятельности.

СПИСОК ЛИТЕРАТУРЫ

[1] T. Kipf , M. Welling. Semi-supervised classification with graph convolutional networks. arXiv:1609.02907, 2016.
[2] Hamilton W. L. Graph representation Learning: учебник / W.L. Hamilton — McGill University, 2020 - 141 с.
[3] Графовые нейронные сети. — URL: https://dyakonov.org/2021/12/30/gnn/ (дата обращения: 15.04.2022). — Текст: электронный.
[4] Understanding Convolution on Graphs. — URL: https://distill.pub/2021/understanding-gnns/ (дата обращения: 15.04.2022). — Текст: электронный.
[5] A Gentle Introduction to Graph Neural Networks. — URL: https://distill.pub/2021/gnn-intro/ (дата обращения: 15.04.2022). — Текст: электронный.
[6] Node Classification on Knowledge Graphs using PyTorch Geometric. — URL: https://www.youtube.com/watch?v=ex2qllcVneY&t=625s&ab_channel=DeepFindr (дата обращения: 15.04.2022). — Текст: электронный.
[7] An Introduction to Graph Neural Network (GNN) For Analysing Structured Data. — URL: https://towardsdatascience.com/an-introduction-to-graph-neural-network-gnn-for-analysing-structured-data-afce79f4cfdc (дата обращения: 15.04.2022). — Текст: электронный.
[8] Graph Neural Networks (GNN) using Pytorch Geometric | Stanford University. — URL: https://www.youtube.com/watch?v=-UjytpbqX4A&ab_channel=LindseyAI (дата обращения: 15.04.2022). — Текст: электронный.
[9] J. Elman. Finding structure in time. Cog. Sci., 14(2):179-211, 1990
[10] I. Goodfellow, Y. Bengio, and A. Courville. Deep Learning. MIT press, 2016.
[11] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola. Deep sets. NeurIPS, 2017
[12] R. Murphy, B. Srinivasan, V. Rao, and B. Ribeiro. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. ICLR, 2018
[13] W.L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. NeurIPS, 2017
[14] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. ICLR, 2015
[15] P. Velickovic, G. Cururull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. ICLR, 2018.

ПРИЛОЖЕНИЯ

```
import torch
torch.__version__
```

```
{"type":"string"}
```

```
try:
  # Check if PyTorch Geometric is installed:
  import torch_geometric
except ImportError:
  # If PyTorch Geometric is not installed, install it.
  %pip install -q torch-scatter -f https://pytorch-geometric.com/whl/torch-1.11.0+cu113.html
  %pip install -q torch-sparse -f https://pytorch-geometric.com/whl/torch-1.11.0+cu113.html
  %pip install -q torch-geometric
```

```
etric (setup.py) ...
```

```
import torch
import torch.optim as optim
import torch.nn.functional as F
from torch import nn
from torch.nn import Linear, LayerNorm
from torch_geometric.nn import GCNConv
from torch_geometric.nn import SAGEConv
from torch_geometric.datasets import Planetoid
import matplotlib.pyplot as plt
from torch_geometric.transforms import NormalizeFeatures
from torch_geometric.utils import accuracy
import numpy as np
```

Dataset investigation

```
# Get some information about dataset

dataset = Planetoid(root='data/Planetoid', name='Cora', transform=NormalizeFeatures())
num_nodes = dataset.data.num_nodes
num_edges = dataset.data.num_edges // 2
train_len = dataset[0].train_mask.sum()
val_len = dataset[0].val_mask.sum()
test_len = dataset[0].test_mask.sum()
other_len = num_nodes - train_len - val_len - test_len
print(f"Dataset: {dataset.name}")
print(f"Num. nodes: {num_nodes} (train={train_len}, val={val_len}, test={test_len},
other={other_len})")
print(f"Num. edges: {num_edges}")
print(f"Num. node features: {dataset.num_node_features}")
```

```
print(f"Num. classes: {dataset.num_classes}")
print(f"Dataset len.: {dataset.len()}")

Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.x
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.tx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.allx
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.y
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.ty
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.ally
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.graph
Downloading https://github.com/kimiyoung/planetoid/raw/master/data/ind.cora.test.index

Dataset: Cora
Num. nodes: 2708 (train=140, val=500, test=1000, other=1068)
Num. edges: 5278
Num. node features: 1433
Num. classes: 7
Dataset len.: 1

Processing...
Done!

# How our nods look like
print(dataset[0].x.shape) # [No. Nodes x Features]

torch.Size([2708, 1433])

# Print some of the normalized word counts of the first datapoint
dataset[0].x[0][:50]

tensor([0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.1111, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
        0.0000, 0.0000, 0.0000, 0.0000, 0.0000])

# How do the labels look like
dataset[0].y

tensor([3, 4, 4,  ..., 3, 3, 3])

# Example of the train mask
dataset[0].train_mask

tensor([ True,  True,  True,  ..., False, False, False])

# Example of the test mask
dataset[0].test_mask
```

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

# Example of the validation mask
dataset[0].val_mask

tensor([False, False, False,  ..., False, False, False])
```

Graph Convolutional Network

```
class GCN(torch.nn.Module):
    def __init__(self, num_node_features, num_classes, hidden_dim=16, dropout_rate = 0.5):
        super().__init__()
        self.dropout1 = torch.nn.Dropout(dropout_rate)
        self.conv1 = GCNConv(num_node_features, hidden_dim)
        self.relu = torch.nn.ReLU(inplace=True)
        self.dropout2 = torch.nn.Dropout(dropout_rate)
        self.conv2 = GCNConv(hidden_dim, num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        # First layer
        x = self.dropout1(x)
        x = self.conv1(x, edge_index)
        x = self.relu(x)

        # Second layer
        x = self.dropout2(x)
        x = self.conv2(x, edge_index)
        #x = self.relu(x)

        return x

print("Graph Convolutional Network (GCN):")
GCN(dataset.num_node_features, dataset.num_classes, 64)

Graph Convolutional Network (GCN):

GCN(
  (dropout1): Dropout(p=0.5, inplace=False)
  (conv1): GCNConv(1433, 64)
  (relu): ReLU(inplace=True)
  (dropout2): Dropout(p=0.5, inplace=False)
  (conv2): GCNConv(64, 7)
)
```

Training and Evaluation

```
def train_step(model, data, optimizer, loss_fn):
    model.train()
```

```
    optimizer.zero_grad()
    mask = data.train_mask

    logits = model(data)[mask]
    y = data.y[mask]
    loss = loss_fn(logits, y)

    preds = logits.argmax(dim=1)
    acc = accuracy(preds, y)

    loss.backward()
    optimizer.step()

    return loss.item(), acc


@torch.no_grad()
def eval_step(model, data, loss_fn, stage):
    model.eval()
    mask = getattr(data, f"{stage}_mask")

    logits = model(data)[mask]
    y = data.y[mask]
    loss = loss_fn(logits, y)

    preds = logits.argmax(dim=1)
    acc = accuracy(preds, y)

    return loss.item(), acc

def train(model, data, optimizer, loss_fn = torch.nn.CrossEntropyLoss(),
    max_epochs = 200, early_stopping = 10, print_interval = 20, verbose = True):

    history = {"loss": [], "val_loss": [], "acc": [], "val_acc": []}
    for epoch in range(max_epochs):
        loss, acc = train_step(model, data, optimizer, loss_fn)
        val_loss, val_acc = eval_step(model, data, loss_fn, "val")
        history["loss"].append(loss)
        history["acc"].append(acc)
        history["val_loss"].append(val_loss)
        history["val_acc"].append(val_acc)
        if epoch > early_stopping and val_loss > np.mean(history["val_loss"][-(early_stopping + 1) :
-1]):
            if verbose:
                print("\nEarly stopping...")

            break

        if verbose and epoch % print_interval == 0:
            print(f"\nEpoch: {epoch}\n----------")
```

```
            print(f"Train loss: {loss:.4f} | Train acc: {acc:.4f}")
            print(f"  Val loss: {val_loss:.4f} |   Val acc: {val_acc:.4f}")

        test_loss, test_acc = eval_step(model, data, loss_fn, "test")
        if verbose:
            print(f"\nEpoch: {epoch}\n----------")
            print(f"Train loss: {loss:.4f} | Train acc: {acc:.4f}")
            print(f"  Val loss: {val_loss:.4f} |   Val acc: {val_acc:.4f}")
            print(f" Test loss: {test_loss:.4f} |  Test acc: {test_acc:.4f}")

        return history

    def plot_history(history, title, font_size = 14):
        plt.figure(figsize=(12, 4))

        plt.suptitle(title, fontsize=font_size)
        ax1 = plt.subplot(121)
        ax1.set_title("Loss")
        ax1.plot(history["loss"], label="train")
        ax1.plot(history["val_loss"], label="val")
        plt.xlabel("Epoch")
        ax1.legend()

        ax2 = plt.subplot(122)
        ax2.set_title("Accuracy")
        ax2.plot(history["acc"], label="train")
        ax2.plot(history["val_acc"], label="val")
        plt.xlabel("Epoch")
        ax2.legend()

    input_dim = dataset.num_features
    HIDDEN_CHANNELS = 64
    OUTPUT_DIM = 7
    DROPOUT = 0.75
    SEED = 42
    MAX_EPOCHS = 200
    LEARNING_RATE = 0.01
    WEIGHT_DECAY = 5e-4
    EARLY_STOPPING = 10

    torch.manual_seed(SEED)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    model = GCN(input_dim, OUTPUT_DIM, HIDDEN_CHANNELS, DROPOUT).to(device)
    data = dataset[0].to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE,
weight_decay=WEIGHT_DECAY)
    history = train(model, data, optimizer, max_epochs=MAX_EPOCHS,
early_stopping=EARLY_STOPPING)
```

```
plot_history(history, "GCN")
```

```
Epoch: 0
----------
Train loss: 1.9450 | Train acc: 0.1500
  Val loss: 1.9408 |   Val acc: 0.3600

Epoch: 20
----------
Train loss: 1.5975 | Train acc: 0.7500
  Val loss: 1.7242 |   Val acc: 0.7700

Epoch: 40
----------
Train loss: 1.0704 | Train acc: 0.8286
  Val loss: 1.3318 |   Val acc: 0.8040

Epoch: 60
----------
Train loss: 0.7739 | Train acc: 0.8929
  Val loss: 1.0373 |   Val acc: 0.8100

Early stopping...

Epoch: 79
----------
Train loss: 0.5608 | Train acc: 0.9214
  Val loss: 0.9312 |   Val acc: 0.8000
 Test loss: 0.8899 |  Test acc: 0.8160
```

```
[]
```