МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет прикладной математики и информатики

Кафедра дискретной математики и алгоритмики

Попов Дмитрий Валерьевич

ГРАФОВЫЕ НЕЙРОННЫЕ СЕТИ И ИХ ПРИМЕНЕНИЕ ДЛЯ РЕШЕНИЯ КОМБИНАТОРНЫХ ЗАДАЧ

Магистерская диссертация

1-31 80 09 «Прикладная математика и информатика»

Научный руководитель
Орлович Юрий Леонидович
кандидат физико-математических наук, доцент

Допущена к защите
«___» _____ 2021 г.
Зав. кафедрой дискретной математики и алгоритмики
_____ В.М. Котов
доктор физико-математических наук, профессор

Минск, 2021

ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ **2**
ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ **4**
АГУЛЬНАЯ ХАРАКТАРЫСТЫКА ПРАЦЫ **5**
ABSTRACT **6**
ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ **7**
ВВЕДЕНИЕ **8**
ГЛАВА 1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ ИЗ ТЕОРИИ ГРАФОВ И НЕЙРОННЫХ СЕТЕЙ **9**
1.1. Основные понятия теории графов 9
1.1.1. Граф 9
1.1.2. Независимые множества 9
1.1.3. Раскраска графа 10
1.2. Основные понятия теории нейронных сетей 11
1.2.1. Искусственные нейронные сети 11
1.2.2. Графовая нейронная сеть 14
1.2.3. Сверточные нейронные сети 16
1.2.4. Графовые сверточные сети 16
1.2.5. Encoder-Decoder 18
ГЛАВА 2. ИСПОЛЬЗОВАННЫЕ МОДЕЛИ ГРАФОВЫХ НЕЙРОННЫХ СЕТЕЙ **20**
2.1. Модели GNN 20
2.1.1. Принципы построения моделей GNN 20
2.1.2. Модель GraphNetwork 22
2.1.3. Модель GraphIndependent 23
2.1.4. Модель MPNN 24
2.1.5. Модель Deep Sets 25
2.1.6. Модель CommNet 26
2.2. Модели GCN 27
2.2.1. Модель GraphConv 27

2.2.2. Модель GraphSAGE 28
2.2.3. Модель SGConv 28
2.2.4. Модель ChebConv 29
2.2.5. Модель GATConv 30
ГЛАВА 3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРОВЕДЕНИЯ ЭКСПЕРИМЕНТОВ 32
3.1. Используемые инструменты 32
3.2. Реализация общих компонент приложения 32
3.2.1. Работа с графовыми данными 33
3.2.2. Генерация случайных величин 34
3.2.3. Остальная функциональность и константы 35
3.3. Реализация компонент для запуска экспериментов над GNN 37
3.3.1. Принцип работы библиотеки graph_nets 37
3.3.2. Необходимые инструменты для реализации 37
3.3.3. Архитектура приложения запуска экспериментов над GNN ... 38
3.4. Реализация компонент для запуска экспериментов над GCN 41
3.4.1. Принцип работы библиотеки dgl 41
3.4.2. Необходимые инструменты для реализации 42
3.4.3. Архитектура приложения запуска экспериментов над GCN 42
ГЛАВА 4. ЗАПУСК И АНАЛИЗ РЕЗУЛЬТАТОВ ЭКСПЕРИМЕНТОВ 45
4.1. Запуск и анализ экспериментов при решении задачи о наибольшем независимом множестве 45
4.2. Запуск и анализ экспериментов при решении задачи о вершинной раскраске графа 48
ЗАКЛЮЧЕНИЕ 49
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 50
ПРИЛОЖЕНИЕ А 52
ПРИЛОЖЕНИЕ Б 53

ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Магистерская диссертация, 58 с., 19 рис., 1 таблица, 14 источников, 2 приложения.

Ключевые слова: ГРАФОВЫЕ НЕЙРОННЫЕ СЕТИ, ГРАФОВЫЕ СВЕРТОЧНЫЕ НЕЙРОННЫЕ СЕТИ, ГРАФОВЫЕ КОМБИНАТОРНЫЕ ЗАДАЧИ.

Объектом исследования являются комбинаторные графовые задачи. Предмет исследования — построение решений рассматриваемых задач с помощью нейронных сетей.

Цель работы: изучить методы решения графовых комбинаторных задач и исследовать применимость нейронных сетей к решению таких задач; изучить и предложить модели нейронных сетей для возможного решения графовых комбинаторных задач; реализовать программу для проведения экспериментов над предложенными моделями; провести эксперименты над предложенными моделями и проанализировать результаты.

Методы, используемые в работе: методы программирования нейронных сетей.

Результаты: результаты проведенных экспериментов над предложенными моделями нейронных сетей; разработанная программа для проведения экспериментов над различными моделями нейронных сетей.

Область применения: научные исследования в области теории графов, нейронных сетей.

АГУЛЬНАЯ ХАРАКТАРЫСТЫКА ПРАЦЫ

Магістарская дысертацыя, 58 с., 19 мал., 1 табл., 14 крыніц, 2 дадатка.

Ключавыя словы: ГРАФАВЫЯ НЕЙРОНАВЫЯ СЕТКІ, ГРАФАВЫЯ СКРУТКАВЫЯ НЕЙРОНАВЫЯ СЕТКІ, ГРАФАВЫЯ КАМБІНАТОРНЫЯ ЗАДАЧЫ.

Аб'ектам даследавання з'яўляюцца камбінаторныя графавыя задачы. Прадмет даследавання — пабудова рашэнняў разгляданых задач з дапамогай нейронавых сетак.

Мэта працы: вывучыць метады рашэння графавых камбінаторных задач і даследаваць дастасавальнасць нейронавых сетак да вырашэння такіх задач; вывучыць і прапанаваць мадэлі нейронавых сетак для магчымага рашэння графавых камбінаторных задач; рэалізаваць праграму для правядзення эксперыментаў над прапанаванымі мадэлямі; правесці эксперыменты над прапанаванымі мадэлямі і прааналізаваць вынікі.

Метады, якія выкарыстоўваюцца ў рабоце: метады праграмавання нейронавых сетак.

Вынікі: вынікі праведзеных эксперыментаў над прапанаванымі мадэлямі нейронавых сетак; распрацаваная праграма для правядзення эксперыментаў над рознымі мадэлямі нейронавых сетак.

Вобласць прымянення: навуковыя даследаванні ў галіне тэорыі графаў, нейронавых сетак.

ABSTRACT

Master thesis, 58 p., 19 fig., 1 table, 14 sources, 2 appendixes.

Keywords: GRAPH NEURAL NETWORKS, GRAPH CONVOLUTIONAL NEURAL NETWORKS, GRAPH COMBINATORIAL PROBLEMS.

The object of research is combinatorial graph problems. The subject of the research is the construction of solutions to considered problems using neural networks.

Purpose of the work: to study methods of solving graph combinatorial problems and to investigate the applicability of neural networks to solving such problems; study and propose models of neural networks for the possible solution of graph combinatorial problems; implement a program for conducting experiments on the proposed models; conduct experiments on the proposed models and analyze the results.

Methods, used in the work: methods of programming neural networks.

Results: the results of experiments conducted on the proposed models of neural networks; the developed program for conducting experiments on various models of neural networks.

Application area: a scientific research in the field of graph theory, neural network.

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

ИНС – искусственная нейронная сеть.
GNN – графовая нейронная сеть (graph neural network).
CNN – сверточная нейронная сеть (convolutional neural network).
GCN – графовая сверточная сеть (graph convolutional network).
MPNN – Message-Passing Neural Network.
SGConv – Simplifying Graph Convolutional Network.
SGConv – Simplifying Graph Convolutional Network.
ChebConv – Chebyshev Spectral Graph Convolution.
GATConv – Graph Attention Network Convolution.

ВВЕДЕНИЕ

Тема магистерской диссертации связана с постановкой и решением комбинаторных задач. Такие задачи часто имеют место в логистической отрасли. Однако данные задачи являются NP-трудными. Это означает, что до сих пор не найден хотя бы один алгоритм решения данных задач, работающий за полиномиальное время. Следовательно, при масштабировании задачи будет значительно увеличиваться время выполнения алгоритма.

Однако, в связи с развитием отрасли искусственного интеллекта, в частности, методов машинного обучения и искусственных нейронных сетей, возникает возможность применить данные методы в теории графов, в том числе, в таких задачах, как задаче о независимом множестве в графе и задаче о вершинной раскраске. При этом ответ на задачу будет являться не точным, а верным с какой-либо вероятностью. Это позволяет пренебречь точностью, однако, значительно выиграть в скорости работы алгоритма.

В первой главе данной диссертации описана теоретическая составляющая, необходимая для решения данных задач. Также введены некоторые понятия из теории искусственных нейронных сетей, использованные в практической составляющей работы. Во второй главе описаны использованные модели и их улучшения для решения указанных комбинаторных задач на графах. В третьей главе описана архитектура и схема построения приложения для проведения экспериментов с использованием выбранных моделей. В четвертой главе описаны результаты данных экспериментов для рассмотренных комбинаторных задач на графах.

ГЛАВА 1. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ ИЗ ТЕОРИИ ГРАФОВ И НЕЙРОННЫХ СЕТЕЙ

1.1. Основные понятия теории графов

1.1.1. Граф

Введем некоторые понятия из теории графов, необходимые для постановки задач.

Определение 1.1.1. *Граф*, или *неориентированный граф*, G — это упорядоченная пара $G = \{ V, E \}$, где V — непустое множество вершин или узлов, а E — множество пар (в случае неориентированного графа — неупорядоченных) вершин, называемых рёбрами [1].

Далее под графом будем иметь в виду неориентированный граф. Ниже находится пример графа (см. рис. 1.1).

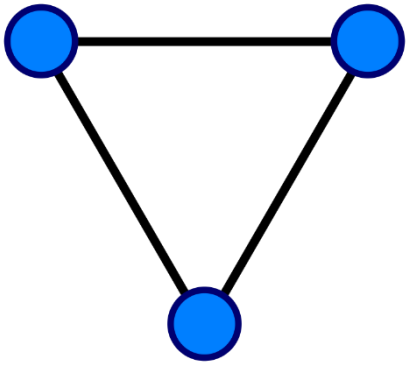

Рисунок 1.1 — Пример графа. Круги являются вершинами, а отрезки, соединяющие их, – ребра

1.1.2. Независимые множества

Рассмотрим задачи, связанные с понятием независимости в графах. Для этого введем следующие определения.

Определение 1.1.2. Множество вершин графа называется *независимым*, если никакие две вершины этого множества не соединены ребром. Другими словами, индуцированный этим множеством подграф состоит из изолированных вершин [2].

Для понимания рассмотрим рисунок ниже (см рис. 1.2):

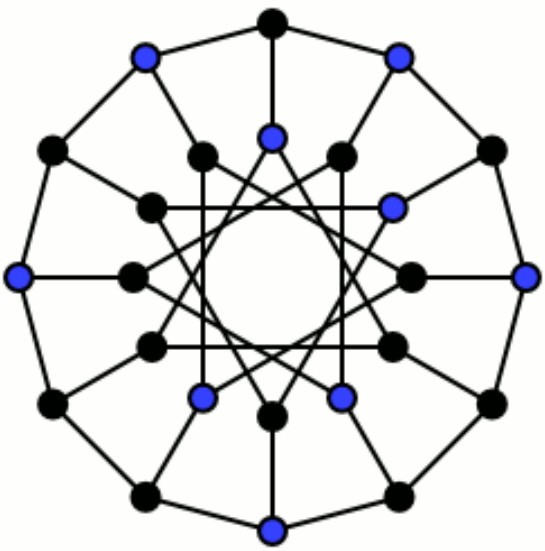

Рисунок 1.2 — Пример независимого множества на графе

На данном рисунке синими вершинами обозначено независимое множество данного графа. Это означает, что никакая синяя вершина не смежна с другой синей вершиной.

Определение 1.1.3. *Максимальным независимым множеством* называется независимое множество, не являющееся подмножеством другого независимого множества.

Определение 1.1.4. Независимое множество наибольшей мощности называется *наибольшим независимым множеством*. Любое наибольшее независимое множество является максимальным, обратное не всегда верно.

Вернувшись к предыдущему примеру, получим, что множество синих вершин также является максимальным независимым множеством. Для доказательства достаточно перебрать все черные вершины, покрасить каждую в синий цвет и проверить выполнение независимости в новом множестве.

Определение 1.1.5. *Числом независимости (α(G))* графа называется мощность наибольшего независимого множества.

Число независимости графа из данного примера равно 9 (как мощность найденного наибольшего независимого множества).

Задача о независимом множестве, формулируется следующим образом: в заданном графе *G* требуется найти независимое множество наибольшего размера [2]. Данная задача является NP-трудной, что означает, что на данный момент для нее еще не найдено полиномиального алгоритма решения.

1.1.3. Раскраска графа

Определение 1.1.6. *Вершинная раскраска* графа — назначение цветовых меток вершинам графа так, чтобы любые две вершины, имеющие общее ребро,

имели разные цвета [3]. В универсальном случае цвет вершины представляет собой некоторое натуральное число.

Определение 1.1.7. Раскраска с использованием k цветов называется *k-раскраской*. Наименьшее число цветов, необходимое для раскраски графа G, называется его *хроматическим числом* и обозначается через $\chi(G)$ [3].

Рассмотрим пример ниже (см. рис. 1.3):

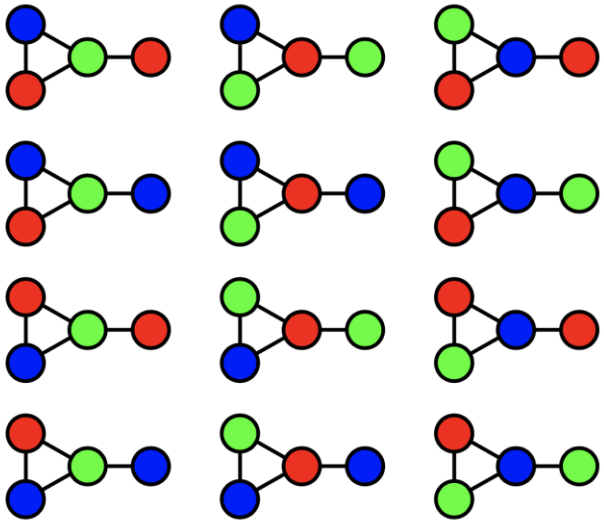

Рисунок 1.3 — 3-раскраска графа на 4 вершинах 12 различными способами

Как показано на примере, существует множество способов раскрасить одну и ту же вершину в пределах одной раскраски. Однако, данные раскраски являются 3-раскрасками, т.к. никакие две соседние вершины в графе не раскрашены одним цветом

Далее рассмотрим основные понятия нейронных сетей.

1.2. Основные понятия теории нейронных сетей

1.2.1. Искусственные нейронные сети

Для понимания принципа работы искусственных нейронных сетей введем некоторые понятия.

Определение 1.2.1. *Искусственная нейронная сеть (ИНС)* — математическая модель, а также её программное или аппаратное воплощение, построенная по принципу организации и функционирования биологических нейронных сетей — сетей нервных клеток живого организма [4].

ИНС представляет собой систему соединённых и взаимодействующих между собой простых процессоров (искусственных нейронов) [4]. Пример нейронной сети представлен ниже (см рис. 1.4):

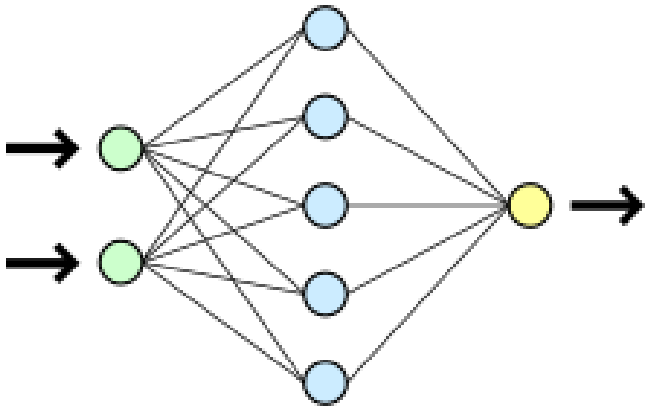

Рисунок 1.4 — Пример нейронной сети

На данном рисунке зеленым цветом обозначены входные нейроны, которые обрабатывают входные данные нейронной сети, голубым — скрытый слой, определяющий внутреннюю логику сети, а желтым — выходные нейроны, обрабатывающие данные со скрытого слоя и выдающие ответ.

В свою очередь, схема одного простейшего искусственного нейрона представлена на следующем рисунке (см рис. 1.5):

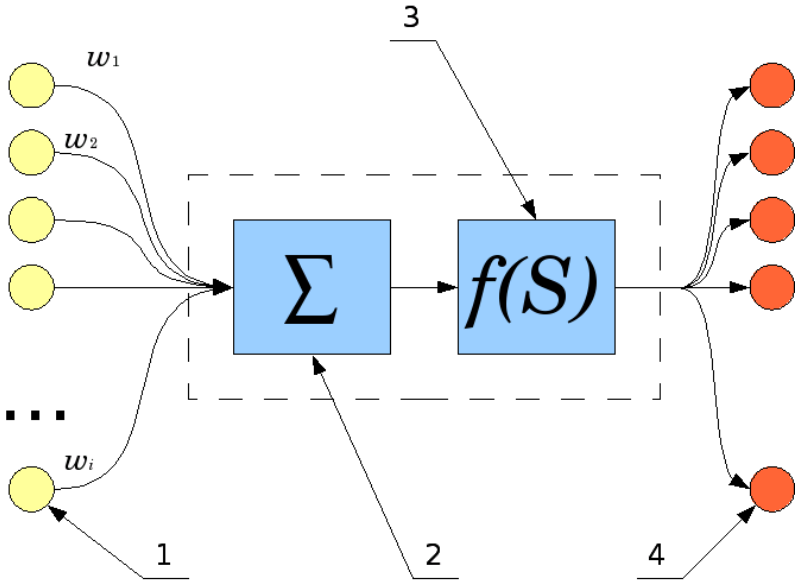

Рисунок 1.5 — Схема простого нейрона. 1 — нейроны, выходные сигналы которых поступают на вход, ω_i — веса входных нейронов, 2 — сумматор, 3 — функция активации нейрона, 4 — нейроны, на входы которых подается выходной сигнал данного [6]

Определение 1.2.2. Математически, *искусственный нейрон* — это взвешенный сумматор, определенный следующим образом:

$$y = f(\sum_{i=1}^{n} \omega_i x_i + \omega_0 x_0), \qquad (1.1)$$

где ω_i — веса входных нейронов, x_i — значения сигналов входных нейронов, функция f — функция активации нейрона, устанавливающаяся в зависимости от типа задачи и, как правило, нелинейна, и y — выходной сигнал (значение) нейрона [5].

Определение 1.2.3. Для измерения точности работы нейронной сети и ее корректировки вводится *функция потерь (ошибки)* — функция, характеризующая некоторые потери при неправильном принятии решения на данных.

Данная функция должна быть напрямую связана с функцией подсчета точности результата и быть ей обратно пропорциональна: чем меньше значение функции потерь, тем выше точность. В частности, одной из самых популярных функций потерь является функция *среднеквадратической ошибки (mean squared error, MSE)*, означающая следующее: чем больше расстояние между векторами полученного ответа и действительного, тем ниже точность данной нейронной сети.

Главной особенностью ИНС является способность «обучаться». Технически — это нахождение весовых коэффициентов входных нейронов для каждого нейрона в зависимости от разницы между действительным и полученным ответами.

Отсюда следует основная характеристика успешно обученной нейронной сети — это минимальное значение функции потерь. Для этого используется т.н. *метод градиентного спуска*, при котором находится локальный минимум функции потерь путем высчитывания антиградиента и движения по нему. Математически данный метод можно представить следующей формулой:

$$x^{j+1} = x^j - \lambda \bigtriangledown F(x^j), \qquad (1.2)$$

где $\bigtriangledown F(x^j)$ — градиент вектора x, λ — параметр, корректирующий шаг метода (скорость обучения, *learning rate*) [6].

Вместе с градиентным спуском используется т.н. *метод обратного распространения ошибки*. Данный метод выполняет подсчет выходных значений нейронной сети и вклад каждой связи между нейронами в данный результат. Затем производится подсчет ошибки каждой связи на основе ее вклада в результат и перерасчет новых весовых коэффициентов.

Существует множество различных классов нейронных сетей. Однако самыми распространенными классами для работы с графовыми структурами данных являются класс *графовых нейронных сетей* (*graph neural network, GNN*), а также, в частности, класс *графовых сверточных нейронных сетей (graph convolutional network)* как модификация класса *сверточных нейронных сетей*

(convolutional neural network, CNN). Данный класс часто используется в задачах распознавания изображений.

1.2.2. Графовая нейронная сеть

Одним из классов нейронных сетей для работы с графами является *графовая нейронная сеть (graph neural network, GNN)*.

На вход данной нейронной сети поступает граф, вершины и ребра которого содержат некоторые данные — *признаки (features)*. Данные признаки формируются в виде векторов одинаковой размерности для всех вершин и для всех ребер. Введем матрицу X_N размера $|V| \times F_V$ — матрицу признаков вершин графа, где $|V|$ — количество вершин в графе, F_V — число вершинных признаков. Также введем матрицу X_E размера $|E| \times F_E$ — матрицу признаков ребер графа, где $|E|$ — количество рёбер в графе, F_E — число реберных признаков.

Используя вершинные и реберные признаки, GNN учится представлять каждую вершину v в виде d-размерного вектора (состояния) h_v, который содержит информацию о соседних вершинах данной вершины, а также о смежных данной вершине ребрах. Данную концепцию можно представить математически в следующем виде:

$$h_v = f\left(x_v, x_{co[v]}, h_{ne[v]}, x_{ne[v]}\right), \tag{1.3}$$

где x_v — признаки текущей вершины v, $x_{co[v]}$ — признаки ребер, инцидентных вершине v, $h_{ne[v]}$ — состояния смежных с v вершин, $x_{ne[v]}$ — признаки смежных с v вершин, а f — некоторая функция, проецирующая входные данные в d-мерное пространство, обновляющая состояние текущей вершины в зависимости от параметров *(локальная функция транзакций)* [7].

При представлении каждой вершины в виде d-мерного вектора нейронная сеть выполняет т.н. *кодирование* вершины, т.е. переход вычислений в другое пространство. При этом при кодировании вершин должны сохраниться пространственные признаки вершин: если вершины в графе находятся на большом расстоянии, то на сравнительно большом расстоянии они должны находиться и в новом пространстве.

Классификация происходит по каждой вершине следующим образом:

$$o_v = g(h_v, x_v), \tag{1.4}$$

где g — некоторая локальная функция вывода, которая классифицирует ответ по состоянию и признакам вершины [7].

Для нахождения состояний $h_v, \forall v \in V$ заменим данное уравнение на итеративно обновляющийся процесс:

$$H^{t+1} = F(H^t, X), \tag{1.5}$$

где H^t — некоторое состояние вершин графа на t-ой итерации обработки графа, X — признаки вершин и ребер графа. Такой процесс называется *передачей сообщений (message passing)*, а F — глобальной функцией транзакций, состоящей из локальных функций, определенных для каждой вершины на каждой итерации [7].

Для лучшего понимания рассмотрим пример (см. рис. 1.6):

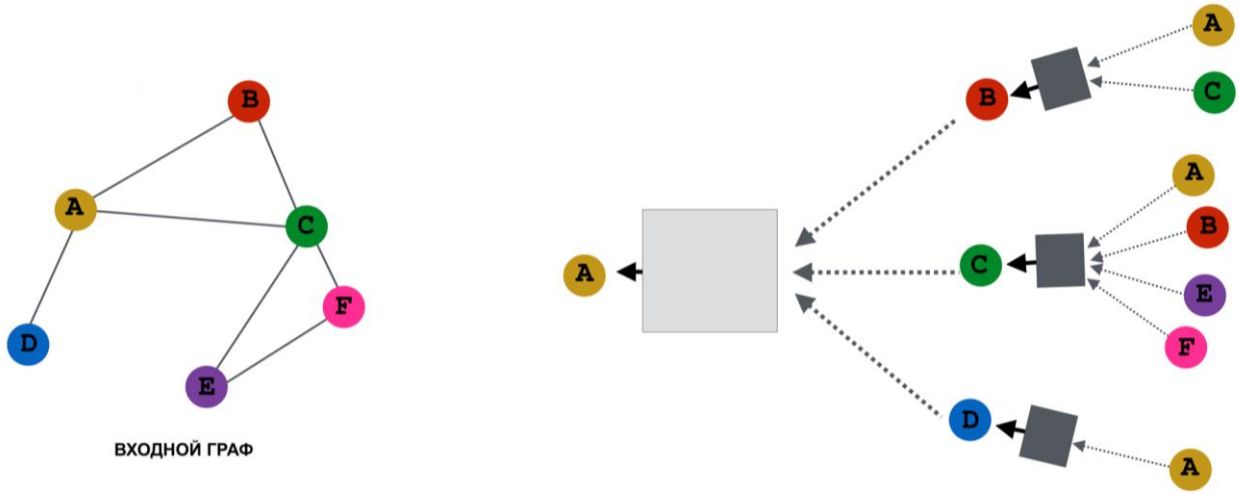

Рисунок 1.6 — Пример кодирования вершины «A» графовой нейронной сетью

На данном примере происходит кодирование вершины «A» с использованием состояний соседних вершин. Как видно на графе слева, вершина «A» смежна с вершинами «B», «C» и «D». Поэтому новое состояние вершины «A» вычисляется в зависимости от состояний этих вершин (серый блок). Однако состояние данных вершин также вычисляется в зависимости от состояний смежных с ними вершин (черный блок). Данные блоки можно интерпретировать как элементы слоя искусственной нейронной сети, соответственно, сами итерации обновления состояния — как слои нейронной сети.

Отсюда возникает проблема графовых нейронных сетей: при обработке достаточно больших графов для достижения приемлемых результатов необходимо обучать достаточно большое количество message-passing слоев. Это значительно увеличивает глубину нейронной сети, а значит, и время ее обучения.

Также вывод классифицированных вершин GNN обобщенно имеет следующий вид:

$$O = G(H, X_N), \tag{1.6}$$

где X_N — все признаки вершин, G — глобальная функция вывода, состоящая из множества локальных функций вывода [7]. В частности, данная функция также может быть искусственной нейронной сетью.

Рассмотрим другие классы нейронных сетей.

1.2.3. Сверточные нейронные сети

Основным принципом работы CNN является свертка входной матрицы в матрицу меньшего размера. Пример свертки можно наблюдать на рисунке ниже (см. рис. 1.7):

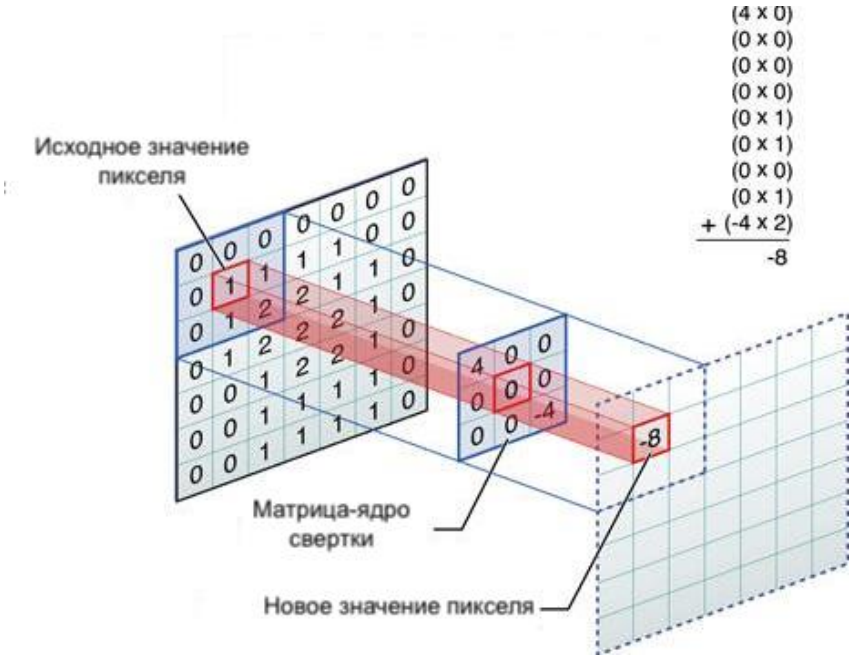

Рисунок 1.7 — Работа свертки в сверточной нейронной сети на примере пиксельного изображения

В данном примере имеется *матрица-ядро* нейронной сети (матрица 3×3). В CNN, как и в простейшей нейронной сети, рассчитывается сумма взвешенных сигналов, но происходит это путем поэлементного умножения матрицы-ядра (матрицы весов) с исходной входной матрицей (матрицей сигналов). После данного действия происходит вычисление суммы элементов получившейся матрицы и сдвиг ядра на следующий элемент. В итоге получаем матрицу меньшего размера.

После выполнения всех преобразований скалярный результат поступает в функцию активации, и возвращается итоговое значение.

Однако напрямую CNN в задачах на графах не используется ввиду неевклидовости графового пространства. Поэтому используется модификация GNN с принципами работы CNN для решения задач на графах — *графовая сверточная сеть (graph convolutional network, GCN)*.

1.2.4. Графовые сверточные сети

Идея данной нейронной сети заключается в применении сверток в неевклидовом графовом пространстве. Для этого, аналогично концепции CNN,

производится подобие свертки, только не в пространственном смысле, а в реляционном: каждая вершина кодируется с помощью смежных вершин.

Рассмотрим пример на рисунке ниже (см. рис. 1.8):

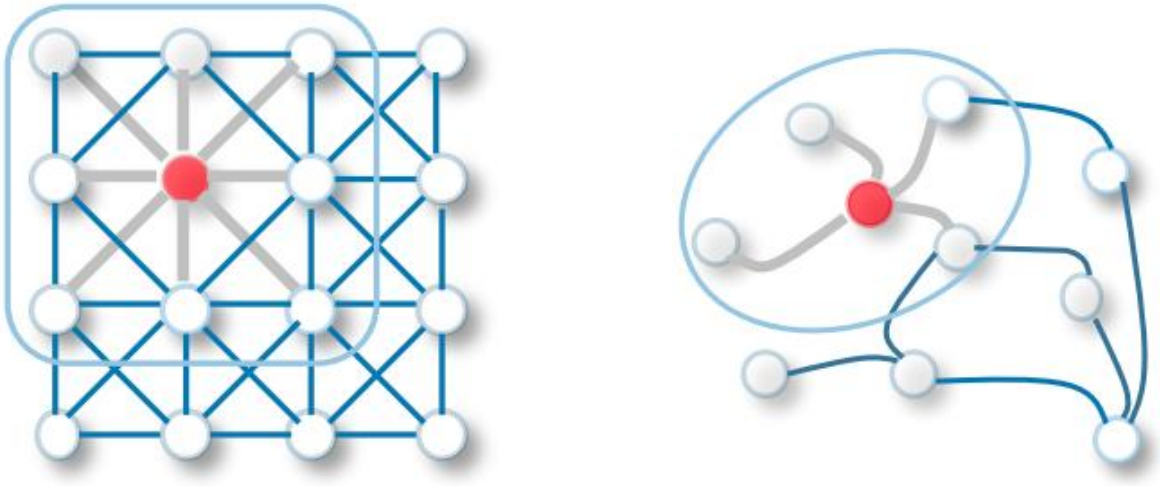

Рисунок 1.8 — Пример различия свертки на изображениях и в графах

Как показано на рисунке слева, изображение можно представить в виде связного графа-сетки пикселей. Соответственно свертка сверточной нейронной сети будет высчитывать взвешенную сумму пикселей-вершин из окружения для каждого пикселя. Но в данной концепции для каждой вершины изначально известно количество ее соседей, а также порядок вычисления взвешенной суммы, т.к. в сверточной нейронной сети происходит умножение матрицы-ядра поэлементно с текущей областью изображения.

При выполнении свертки на рисунке справа происходит та же свертка, но с несколькими отличиями. Изначально на этапе компиляции для каждой вершины неизвестно количество ее соседей, а также их порядок, что не позволяет использовать подобие матриц в данном пространстве.

Для решения данной задачи можно заменить свертку на несколько иных операций: *агрегации* и *обновления*. Первая операция агрегирует данные признаков соседей и текущей вершины. Вторая операция выполняет кодирование данной вершины на основе результата агрегации. В итоге при выполнении данных операций получается новое значение состояния вершины.

Стоит отметить, что при данном подходе свертка не уменьшает размерность графа (число вершин графа), в отличие от сверток CNN. Кодирование каждой вершины выполняется последовательно и зависит от предыдущего состояния соседей и признаков, что не может уменьшать размерность вычислений.

Очевидно, что в данной концепции, в отличие от GNN, входными данными для GCN являются только матрица вершинных признаков, т.к. агрегация происходит только исходя из вершинных признаков и состояний.

Скрытый слой GCN может быть записан в виде:

$$H_i = f(H_{i-1}, A), \tag{1.7}$$

где $H_0 = X$— матрица вершинных признаков, A — матрица смежности, f — функция перехода, а каждый уровень H_i представляет собой вектор состояний вершин графа [8]. По аналогии с GNN, функция f является функцией глобальной функцией транзакций и представляет собой процесс передачи сообщений.

В простейшем случае:

$$f(H_i, A) = \sigma(AH_iW_i), \tag{1.8}$$

где W_i — матрица весов уровня i, являющиеся фильтрующими параметрами, а σ — функция активации (например, ReLU или сигмоида). При умножении AH_i каждый ряд данной матрицы равен сумме признаков соседних вершин, т.е. каждая вершина представима в виде агрегатора собственных соседей. Отсюда возникает проблема, что новое состояние данной вершины никак не зависит от собственного признака, т.к. данная вершина не смежна сама с собой. Для этого вместо матрицы A будем рассматривать матрицу $\hat{A} = A + I$, где I — единичная матрица. Этот процесс называется *самозацикливанием (self-looping)*, т.е. добавление ребра из текущей вершины в текущую.

Также желательно нормализовывать признаки из-за чувствительности градиентного спуска к ним. Данная нормализация может быть произведена умножением $\hat{A}$ на обратную матрицу степеней вершин D^{-1}, где D — диагональная матрица из степеней вершин.

В итоге получим следующую формулу скрытого слоя GCN:

$$H_i = \sigma(D^{-1/2}\hat{A}D^{-1/2}H_{i-1}W_{i-1}). \tag{1.9}$$

Помимо выше описанных классов нейронных сетей, в данной работе использовался также *Encoder-Decoder* подход к обучению. Далее рассмотрим данный подход.

1.2.5. Encoder-Decoder

Данный подход является универсальным подходом в кодировании входных данных в некоторое ограниченное пространство постоянного размера. Модель Encoder-Decoder, в отличие от GNN и GCN, стремится закодировать данные независимо друг от друга в векторное пространство другого размера и раскодировать данные из него. Как правило, данная модель является некоторым

подобием обертки над ядром построенной модели и позволяет перевести вычисления в векторное пространство меньшего размера, что улучшает временные показатели в обучении.

Как видно из названия, данная модель состоит из двух компонент: *энкодера* и *декодера*. Первая компонента призвана перевести вычисление в меньшее векторное пространство. Вторая же выполняет перевод вычислений скрытых слоев нейронной сети в векторное пространство ответа.

Данная концепция широко используется в языковых моделях для решения задачи, например, перевода фразы на другой язык. Однако также данная модель часто используется и при решении задач на графах. Например, при использовании графов с достаточно большим количеством вершинных признаков время, необходимое на обучение нейронной сети, стремительно возрастет. Чтобы уменьшить размерность векторного пространства вершинных признаков, к данной модели добавляют дополнительные слои энкодера и декодера. В итоге скрытые слои будут работать с пространством меньшего размера, что помогает сократить время обучения.

Далее рассмотрим модели, выбранные для решения задач о независимом множестве и вершинной раскраске.

ГЛАВА 2. ИСПОЛЬЗОВАННЫЕ МОДЕЛИ ГРАФОВЫХ НЕЙРОННЫХ СЕТЕЙ

В данной работе использовались различные модели классов GNN и GCN. Рассмотрим данные модели подробнее с теоретической стороны, а также сильные и слабые стороны этих моделей.

2.1. Модели GNN

2.1.1. Принципы построения моделей GNN

Как было описано в п.1.2.3, графовые нейронные сети выполняют кодирование вершин, основываясь на признаках собственной и соседних вершин, а также на реберных признаках. Однако данная концепция не имела под собой конкретики, и до сих пор непонятно, как это практически реализовано. Также данная концепция не предусматривала на ранних этапах решение задачи классификации ребер и задачи классификации графов в целом, что в принципе теряет смысл ее использовать в этих задачах.

Для решения данных проблем введем понятие *блока* — некоторый модуль, принимающий граф в виде входных данных, выполняющий некоторые вычисления над ним и возвращающий новый граф при выводе [9]. Данное понятие позволяет:

- ввести обработку не только вершинных, но и реберных, и глобальных признаков графа;
- унифицировать и представить обработку графа как цепочку последовательных блоков, обрабатывающих различные признаки;
- конфигурировать каждый блок независимо от другого блока.

Исходя из определения и возможностей использования блока, данные блоки можно разделить на 3 типа: *вершинные, реберные* и *глобальные*. Данные блоки различаются в зависимости от области обработки графа: вершинные признаки, реберные и глобальные соответственно. Однако данное разделение не означает, что обработка тех или иных компонент графа (вершин, ребер или глобальных переменных) производится независимо друг от друга. Т.е., например, при обработке вершинных признаков могут использоваться как реберные, так и глобальные признаки. Но результат обработки графа вершинным

блоком должен изменить только вершинные признаки и не должен затрагивать какие-либо иные.

Аналогично с GCN, в блоках вводятся функции агрегации и обновления. При этом наличие функции агрегации в блоке необязательно, т.к. некоторые признаки (например, реберные) могут не зависеть друг от друга. Перейдем от неориентированных графов к ориентированным путем дублирования ребер с получением пар ребер с противоположными направлениями. Определим данные функции ниже:

$$e'_k = \varphi^e(e_k, v_{r_k}, v_{s_k}, u), k = 1, \ldots, |E|, \quad (2.1)$$

$$E'_i = \{(e'_k, r_k, s_k)\}_{r_k=i, k=1:|E|}, i = 1, \ldots, |V|, \quad (2.2)$$

$$\bar{e}_i = \rho^{e\to v}(E'_i), i = 1, \ldots, |V|, \quad (2.3)$$

$$v'_i = \varphi^v(\bar{e}_i, v_i, u), i = 1, \ldots, |V|, \quad (2.4)$$

$$E' = \{(e'_k, r_k, s_k)\}_{k=1:|E|}, \quad (2.5)$$

$$\bar{e}' = \rho^{e\to u}(E'), \quad (2.6)$$

$$V' = \{v'_i\}_{i=1:|V|}, \quad (2.7)$$

$$\bar{v}' = \rho^{v\to u}(V'), \quad (2.8)$$

$$u' = \varphi^u(\bar{e}', \bar{v}', u), \quad (2.9)$$

где e, v, u обозначают ребро, вершину и глобальную компоненту соответственно; E и V — множества ребер и вершин ориентированного графа соответственно; φ^e, φ^v, φ^u — функции обновления состояний ребер, вершин и глобальной компоненты соответственно; $\rho^{e\to v}$, $\rho^{e\to u}$, $\rho^{v\to u}$ — функции агрегирования ребер относительно каждой вершины, ребер глобально и вершин глобально соответственно; r_k и s_k — индексы вершин получателя и отправителя относительно k-ого ребра соответственно [9].

Следует также отметить, что формула (2.1) является единственным элементом реберного блока, формулы (2.2)-(2.4) выполняются в одном цикле и относятся к вершинному блоку, а формулы (2.5)-(2.9) относятся к глобальному блоку. Отсюда следует вывод о порядке обработки данных блоков: сначала обрабатываются данные в реберном блоке, затем результат данной обработки идет в вершинный блок, после чего производятся вычисления глобального блока.

Реализация данных функций обновления состояний и агрегирования полностью зависит от выбранной модели. В некоторых моделях данные функции могут быть реализованы с помощью различных видов нейронных сетей, в

других — с помощью простейших функций агрегирования: суммирования, максимума-минимума, среднего и т.д.

Также в зависимости от конкретной модели некоторые компоненты могут не использоваться вовсе. Например, существуют модели, ответ которых не зависит от признаков вершин. Следовательно, нет смысла агрегировать вершинные признаки, а также обновлять их состояния. Для этого в обобщенной блочной модели данные блоки имеют линейные функции агрегации и обновления состояния: значение данной функции равно ее параметру.

Далее рассмотрим некоторые модели, выбранные для проведения экспериментов в задачах независимости и вершинной раскраски в графах.

2.1.2. Модель GraphNetwork

Как описано в предыдущем пункте, каждая модель может состоять из 3 связанных между собой видов блоков: реберный, вершинный и глобальный.

Модель *GraphNetwork* представляет собой связанную всеми возможными способами схему соединения всех типов блоков между собой. Данная схема показана на рисунке ниже (см. рис. 2.1):

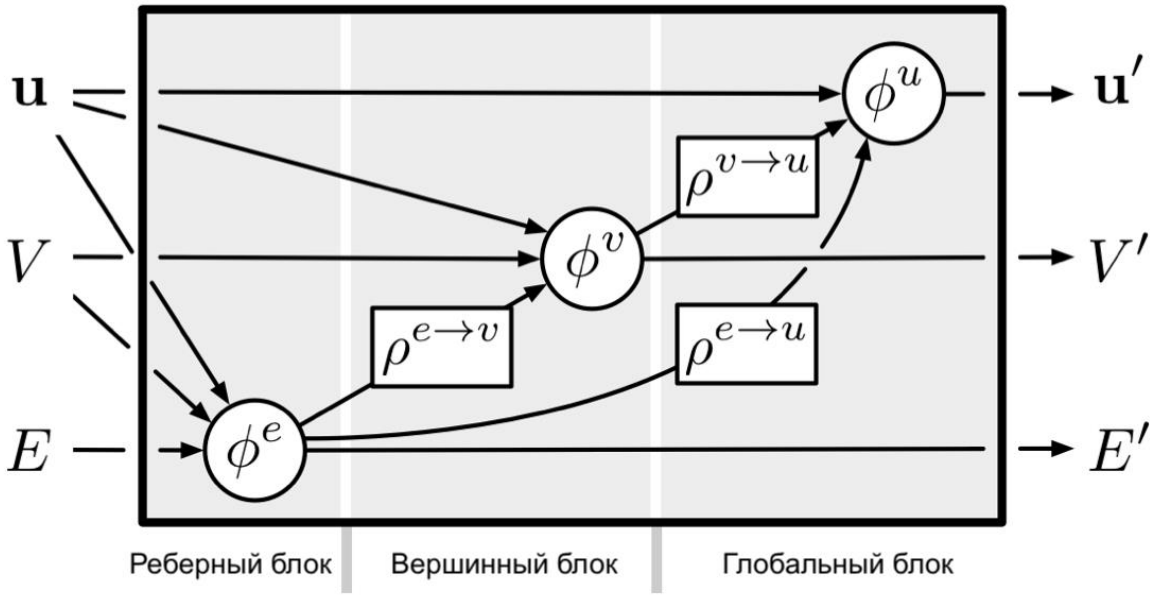

Рисунок 2.1 — Схема работы модели GraphNetwork

Данная полносвязная модель имеет главное преимущество: связь всех компонент между собой дает понимание каждой компоненте об общей картине графа в целом. Это дает возможность работать с более сложными задачами на графах с наличием информации о всех компонентах и получать достаточно неплохой результат.

Наряду с этим, есть несколько важных недостатков. Во-первых, время обучения также увеличится из-за наличия абсолютно всех связей между

блоками. Во-вторых, при неиспользовании различных компонент (например, глобальных признаков) следует занулить данные признаки, при этом это может усложнить обучение данной модели из-за наличия нулей, что также увеличит время обучения. В некоторых случаях данная модель будет избыточна и не принесет какого-либо преимущества.

Однако, данная модель является стандартом среди моделей GNN. Определим функции агрегации и обновления состояния для данной модели.

$$\varphi^e(e_k, v_{r_k}, v_{s_k}, u) = NN_e([e_k, v_{r_k}, v_{s_k}, u]), \tag{2.10}$$

$$\varphi^v(\bar{e}_i, v_i, u) = NN_v([\bar{e}_i, v_i, u]), \tag{2.11}$$

$$\varphi^u(\bar{e}', \bar{v}', u) = NN_g([\bar{e}', \bar{v}', u]), \tag{2.12}$$

$$\rho^{e\to v}(E'_i) = \sum_{\{k:r_k=i\}} e'_k, \tag{2.13}$$

$$\rho^{e\to u}(E') = \sum_k e'_k, \tag{2.14}$$

$$\rho^{v\to u}(V') = \sum_i v'_i, \tag{2.15}$$

где NN_e, NN_v, NN_u — различные нейронные сети, определенные для каждого типа блока, а их параметры — конкатенированные данные [9]. Также функции агрегирования могут быть определены иным образом (например, среднее, максимум-минимум), но для общности определим их как суммы. Далее также будут использоваться данные определения при определении различных моделей.

2.1.3. Модель GraphIndependent

Модель *GraphIndependent* представляет собой независимое кодирование признаков друг от друга. Рассмотрим схему данной модели ниже (см. рис. 2.2):

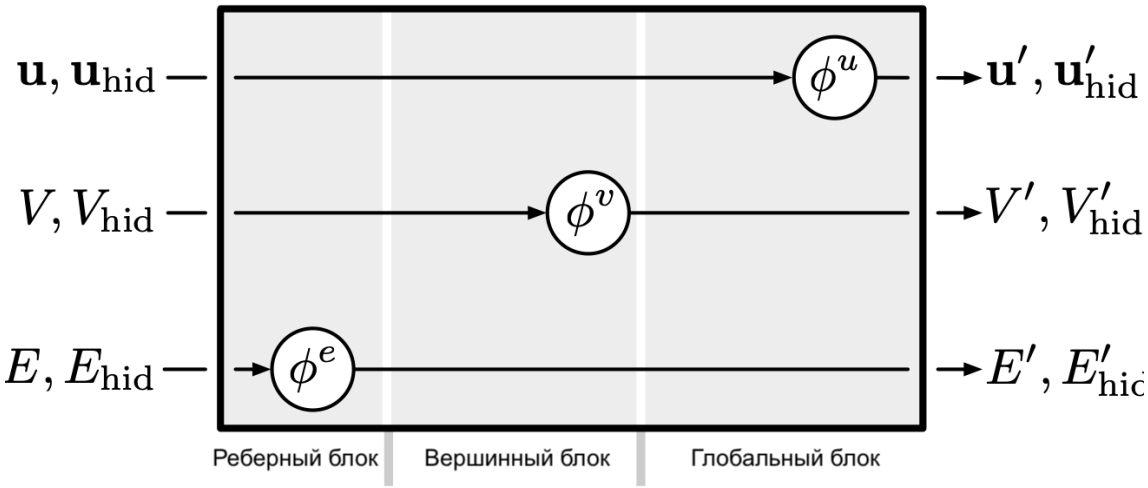

Рисунок 2.2 — Схема работы модели GraphIndependent

На данном рисунке каждый блок кодирует собственную компоненту независимо друг от друга. Кроме этого, также можно произвести настройку с сохранением скрытого состояния ($u_{hid}, V_{hid}, E_{hid}$). Для этого в качестве функций обновления можно определить рекуррентную нейронную сеть с эффектом памяти. Это позволит обновлять, помимо нового значения, контекст окружения. Данная технология часто используется при обработке текстов для сохранения контекста.

В графовых нейронных сетях данная модель чаще всего используется в качестве энкодера-декодера. Поместив данную модель перед и после основных вычислений, можно получить сокращение размеров векторных пространств глобальных, вершинных и реберных признаков.

2.1.4. Модель MPNN

Изначально модель *Message-Passing Neural Network (MPNN)* подразумевает собой разделение обработки графа на 2 фазы: *message-passing* и *readout*. Первая фаза напоминает уже известный *message passing*, состоящий из агрегации и обновления состояния. Вторая фаза высчитывает вектор глобальных признаков на основе состояний вершин графа. Математически данную модель можно представить следующим образом:

$$m_v^{t+1} = \sum_{w \in N(v)} M^t(h_v^t, h_w^t, e_{vw}), \tag{2.16}$$

$$h_v^{t+1} = U^t(h_v^t, m_v^{t+1}), \tag{2.17}$$

$$u = R(\{h_v^t, v \in G\}), \tag{2.18}$$

где формулы (2.16)-(2.18) показывают соответственно агрегацию признаков, обновление состояний и подсчет вектора глобальных признаков; M^t, U^t, R — функции агрегации, обновления и подсчета глобальных признаков соответственно; e_{vw} — признак ребра между вершинами v и w[10].

Следовательно, если спроецировать данную модель на блочную концепцию, получим следующие утверждения:

- функция M^t играет роль φ^e, но без зависимости от u;
- поэлементное суммирование из (1) можно заменить $\rho^{e \to v}$;
- функция U^t играет роль φ^v;
- функция R — ничто иное как φ^u, но без зависимости от u и E', что в принципе убирает определение $\rho^{e \to u}$ для данной модели [9].

Объединив все выше описанное, получим схему MPNN. Рассмотрим данную схему на следующем рисунке (см. рис. 2.3):

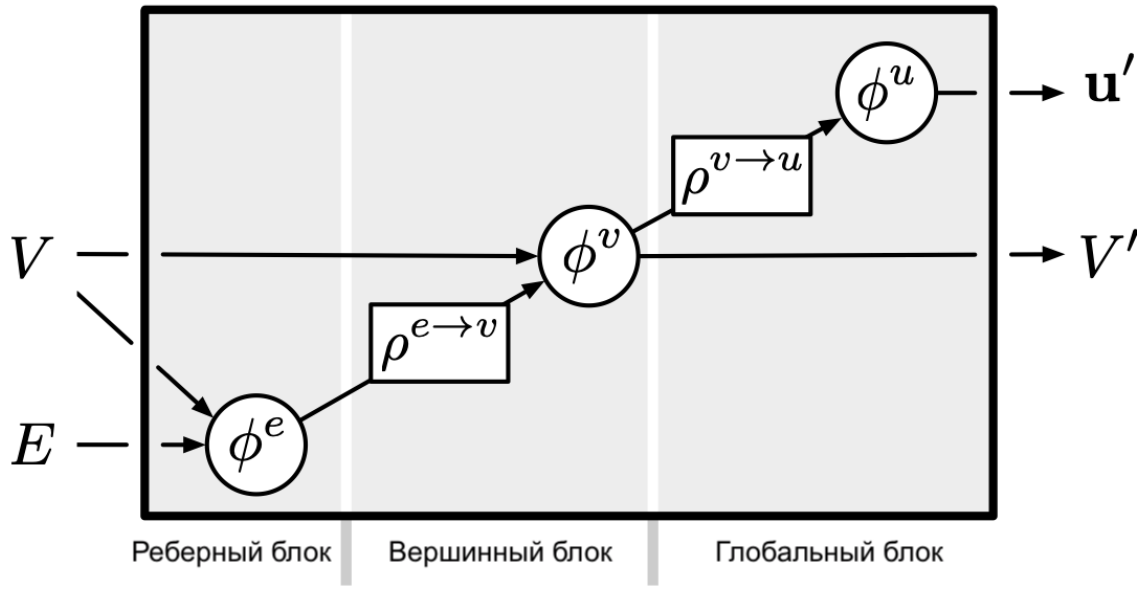

Рисунок 2.3 — Схема модели MPNN

Как было сказано ранее, главными отличиями данной модели от GraphNetwork является отсутствие связи функций обновления состояний ребер и вершин с глобальными признаками графа. Также отсутствует агрегация ребер глобально по графу, что может привести к меньшей чувствительности изменений глобального состояния относительно изменений состояний ребер. Это может дать прирост к скорости вычислений для более простых задач без глобальных и реберных признаков.

2.1.5. Модель Deep Sets

Модель *Deep Sets* оперирует графами как множествами вершин, что может быть интерпретировано как граф без ребер. Данная модель не выполняет никакой обработки над ребрами и не рассматривает ребра в целом.

Рассмотрим схему блоков данной модели (см. рис. 2.4):

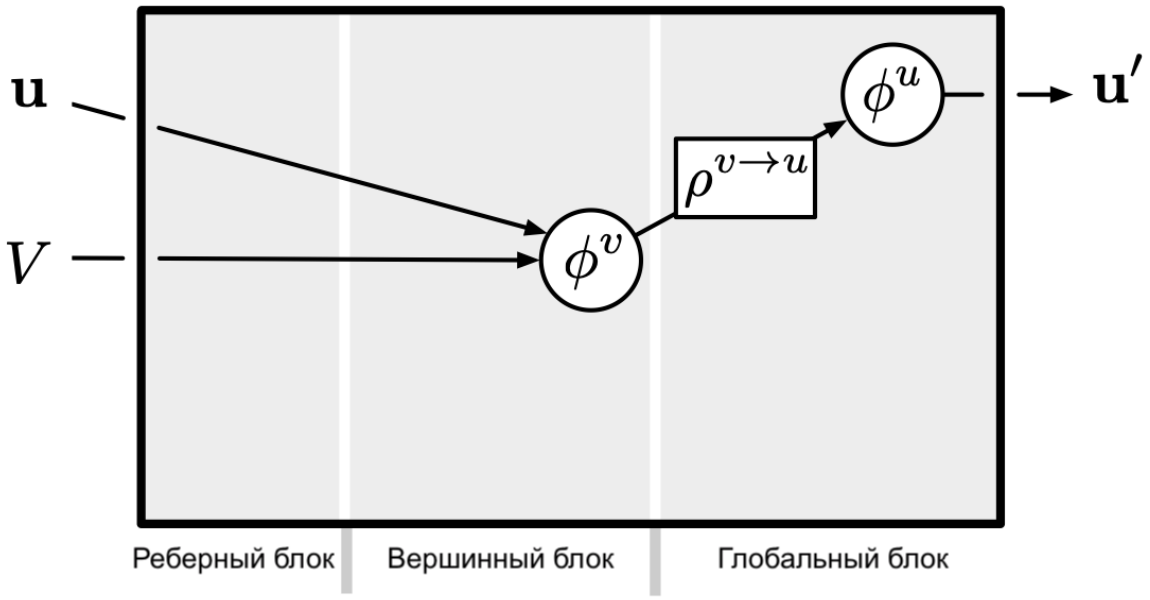

Рисунок 2.4 — Схема модели Deep Sets

Преимуществом данной модели является отказ от обработки ребер, что значительно уменьшает время обучения. Вместе с тем данная модель теряет связи между вершинами и обрабатывает каждую вершину независимо друг от друга, что может значительно ухудшить качество ответа в некоторых задачах.

Данная модель часто используется для вычисления глобальных метрик графа. Например, она выдает достаточно неплохие результаты в задачах классификации графов.

2.1.6. Модель CommNet

Модель *CommNet* базируется на представлении признаков ребер в зависимости от признаков вершин, после чего на основе этих признаков создаются новые вершинные состояния. Математически данную модель можно представить в следующем виде:

$$\varphi^e(e_k, v_{r_k}, v_{s_k}, u) = NN_e(v_{s_k}), \tag{2.19}$$

$$\rho^{e\to v}(E'_i) = \frac{1}{E'_i}\sum_{\{k:r_k=i\}} e'_k, \tag{2.20}$$

$$\varphi^v(\bar{e}_i, v_i, u) = NN_v([\bar{e}_i, NN_{v'}(v_i)]), \tag{2.21}$$

где $NN_{v'}$ — дополнительная нейронная сеть для кодирования исходных признаков вершин [9].

Как можно заметить, в данной модели нет зависимости от глобальных признаков. Также никак не учитываются реберные признаки, более того, они кодируются на основе вершин-отправителей. Это позволяет работать с

признаками вершин, при этом не теряя связности их друг с другом. Поэтому данный метод хорошо себя проявляет в задачах классификации вершин.

Далее рассмотрим модели GCN.

2.2. Модели GCN

2.2.1. Модель GraphConv

Как было сказано ранее, графовые сверточные нейронные сети состоят из двух операций: агрегации вершин и обновления состояний. Однако выбор конкретных операций различается от модели к модели. Также стоит напомнить, что состояния определены только для вершин.

Модель *GraphConv* очень схожа с обычной матричной сверткой. В основе данной модели лежит обновление вершинных состояний в зависимости от весовых параметров. Математически данная графовая свертка выглядит следующим образом:

$$h_i^{l+1} = \sigma(b^l + \sum_{j \in N(i)} \frac{1}{c_{ij}} h_j^l W^l), \tag{2.22}$$

$$c_{ij} = \sqrt{|N_i||N_j|}, \tag{2.23}$$

где σ — функция активации, W — матрица весов, h_i — состояние i-ой вершины, c_{ij} — коэффициент нормировки, b — смещение [11]. Данную формулу можно разделить на агрегацию и обновление состояния соответственно:

$$N_i^l = \sum_{j \in N(i)} \frac{1}{c_{ij}} h_j^l W^l, \tag{2.24}$$

$$h_i^{l+1} = \sigma(b^l + N_i^l), \tag{2.25}$$

где N_i — агрегатор состояний соседних вершин.

Один из недостатков данной модели заключается в том, что она не учитывает реберные признаки. Это означает, что данную модель не стоит использовать в задачах, где они присутствуют. Также видно, что состояние каждой вершины зависит от состояний вершин, соседних с ней, т.е. происходит кодирование текущей вершины с помощью только соседних, и текущее состояние вершины никак не учитывается в результате. Это может негативно сказаться на точности результатов.

2.2.2. Модель GraphSAGE

Модель *GraphSAGE* несколько отличается от предыдущей модели. Первым важным отличием является определение функции агрегации конфигурируемой для каждого слоя по отдельности. К примеру, при накладывании слоев друг на друга на одних слоях можно установить функцию агрегации в виде суммы, на других — в виде среднего. Вторым отличием является нормировка обновленных состояний. Третьим отличием является зависимость функции обновления состояния не только от состояний соседних вершин, но и от текущей вершины.

Математически данную модель можно представить в следующем виде:

$$h_{N(i)}^{l+1} = AGGREGATE_l(\{h_j^l, \forall j \in N(i)\}), \tag{2.26}$$

$$h_i^{l+1} = \sigma(W \cdot [h_i^l, h_{N(i)}^{l+1}]), \tag{2.27}$$

$$h_i^{l+1} = norm(h_i^l), \tag{2.28}$$

где $norm$ — функция нормировки значения нового состояния, $AGGREGATE_l$ — функция агрегации признаков соседних вершин [12].

Исходя из отличий, можно выделить некоторые преимущества данной модели. Во-первых, из-за возможности конфигурирования функций агрегации для каждого слоя данную модель можно подстроить под задачу наилучшим способом, проведя анализ задачи и подходов к ней и эксперименты с теми или иными функциями агрегации и подобрав ту конфигурацию, на которой получился наилучший результат. Во-вторых, нормировка обновленных состояний не даст возможности получить огромные значения в тех или иных вершинах, что исключит возможность переобучения. В-третьих, зависимость функции обновления от текущего состояния даст возможность не потерять связь нового состояния с прошлым, а также не потерять признак текущей вершины при обновлении состояний, что также может сыграть на точности вычислений.

Данная модель является одной из самых популярных моделей в задачах на графах в связи с неплохой точностью результатов.

2.2.3. Модель SGConv

Модель *Simplifying Graph Convolutional Network (SGConv)* имеет под собой идею упростить модель GCN, избавившись от ее чрезмерной усложненности, обусловленной нелинейностью в GCN. Также результирующая нелинейная функция заменена на линейную для уменьшения числа параметров и упрощения модели.

Данные отличия можно наблюдать на следующем рисунке (см. рис. 2.5):

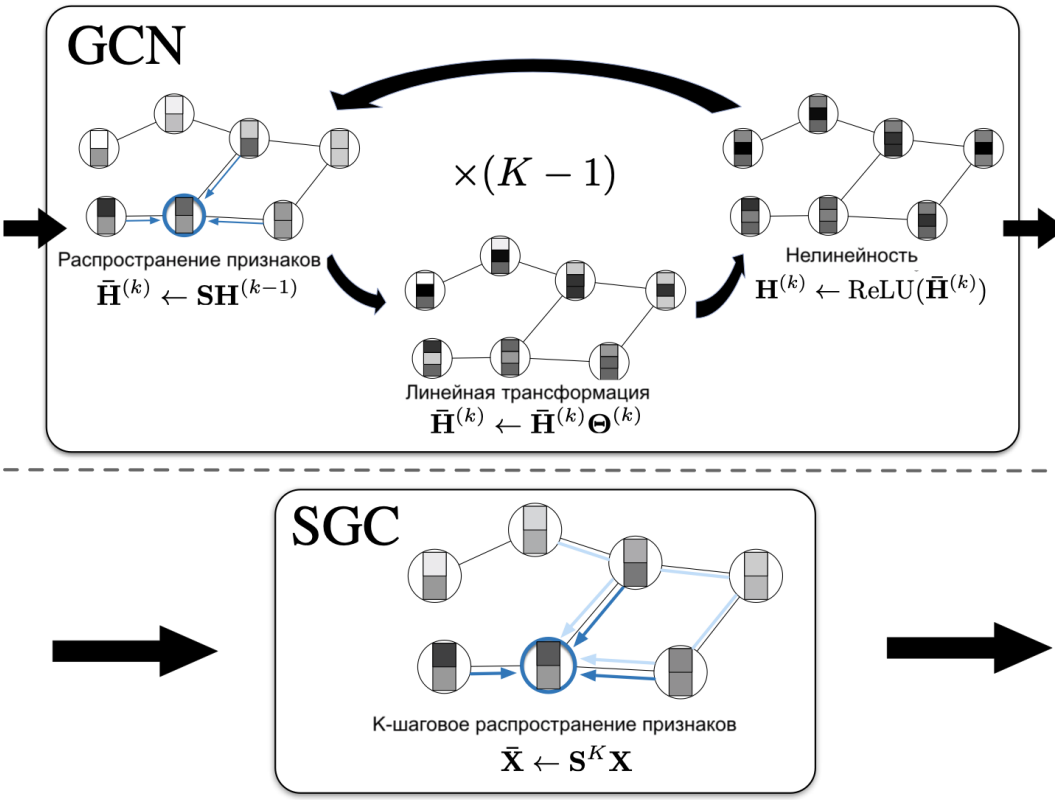

Рисунок 2.1.5 — Различия GraphConv и SGConv

На данном рисунке $S = \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$, где $\hat{A} = A + I$, а $\hat{D} = D + +I$ — диагональная матрица степеней вершин после зацикливания. Также можно заметить, что исчезли линейная и нелинейная трансформация в GCN.

Математически данная модель представляется в следующем виде:

$$H^k = (\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2})^k XW = S^k XW, \tag{2.29}$$

где X — матрица вершинных признаков признаков, а степень при S — степень данной матрицы [11].

Как видим, данная модель сильно упрощена, что явно уменьшит время обучения. Однако вместе с этим есть вероятность, что в более сложных задачах она не сможет как следует обучиться, что приведет к огромным потерям точности.

2.2.4. Модель ChebConv

Модель *Chebyshev Spectral Graph Convolution (ChebConv)* имеет несколько иной подход к обучению. При агрегации вместо использования предыдущих состояний используется значение полинома Чебышева *k*-ого порядка на основе предыдущего состояния. Математически данная свертка представляется следующим образом:

$$h_i^{l+1} = \sum_{k=0}^{K-1} W^{k,l} z_i^{k,l}, \quad (2.30)$$

$$Z^{0,l} = H^l, \quad (2.31)$$

$$Z^{1,l} = \hat{L} \cdot H^l, \quad (2.32)$$

$$Z^{k,l} = 2 \cdot \hat{L} \cdot Z^{k-1,l} - Z^{k-2,l}, \quad (2.33)$$

$$\hat{L} = 2\,(I - \hat{D}^{-1/2} \hat{A} \hat{D}^{-1/2})/\lambda_{max} - I, \quad (2.34)$$

где $\hat{L}$ — нормированный Лапласиан, $Z^{k,l}$ — полином Чебышева *k*-ого порядка [11].

Данная модель может найти некоторые связи в логике задачи среди спектральных характеристик, что в принципе имеет совсем другой подход к решению задачи.

2.2.5. Модель GATConv

Модель *Graph Attention Network Convolution (GATConv)* основывается на концепции *Attention*. Данная концепция основывается на увеличении внимания некоторых деталей данных, т.е. сообщить нейронам из нейронной сети вероятность того или иного исхода в зависимости от состояния нейронов и поступающих на вход данных [13]. Attention сам выявляет факторы, на которые следует обратить внимание, чтобы уменьшить ошибку результата. Данный механизм формирует дополнительную матрицу весов важности, которая символизирует вероятности различных компоненты в той или иной ситуации.

Математически данную модель можно представить в следующем виде:

$$h_i^{l+1} = \sum_{j \in N(i)} \alpha_{i,j} W^l h_j^l, \quad (2.35)$$

$$\alpha_{i,j}^l = softmax(s_{i,j}^l) = \frac{exp(s_{i,j}^l)}{\sum_{k \in N(i)} exp(s_{i,k}^l)}, \quad (2.36)$$

$$s_{i,j}^l = LeakyReLU(\vec{a}^T [W h_i, W h_j]), \quad (2.37)$$

где $\alpha_{i,j}^l$ — результат нелинейной функции *softmax* над attention-значением между вершинами i и j на уровне l, $s_{i,j}^l$ — attention-значение между двумя вершинами i и j на уровне l, $\vec{a}^T$ — транспонированный параметризованный весовой вектор attention [11].

Данная модель принимает в себя все достоинства GCN, при этом также имеет дополнительную attention возможность, что может сыграть большую роль при решении некоторых задач.

Далее рассмотрим программную реализацию запуска экспериментов с описанными моделями.

ГЛАВА 3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРОВЕДЕНИЯ ЭКСПЕРИМЕНТОВ

3.1. Используемые инструменты

Для решения задач о независимости и вершинной раскраски в графах с помощью нейронных сетей в данном случае используются следующие инструменты:

- язык программирования *Python* и его внутренние пакеты;
- фреймворк *Tensorflow v1.x* и *v2* (для работы с процессом обучения нейронных сетей);
- пакеты *networkx, dwave_networkx* и *igraph* (для работы с графовыми данными);
- библиотеки *graph_nets* и *Deep Graph Library (dgl)* (для построения и использования моделей GNN и GCN соответственно);
- фреймворк *sonnet* (для конструирования и кастомизации блоков моделей GNN);
- пакеты *pandas* и *numpy* (для работы с массивами).

Далее рассмотрим программные реализации GNN и GCN для проведения экспериментов. Т.к. данные модели были разработаны с помощью различных несовместимых между собой библиотек, описание программной реализации можно условно разбить на 3 части: реализация компонент для работы с экспериментами над GNN, реализация компонент для работы с GCN и реализация общих компонент, используемых обеими библиотеками.

Полный код описанной ниже реализации находится по ссылке в Приложении А.

3.2. Реализация общих компонент приложения

Общие компоненты данного приложения реализуют, как правило, предобработку графов, генерацию данных, а также различные распределения случайных величин. Помимо этого, также сюда можно отнести различные пользовательские типы данных для настройки конфигурации моделей и процесса

обучения, а также некоторые константы. Данная общая функциональность находится в папке *utils* и затрагивает большинство находящихся там файлов.

3.2.1. Работа с графовыми данными

Для удобной работы с графовыми данными, а также генерации случайных графов создан файл *utils/graph.py*. В данном файле реализована функциональность получения всех наибольших независимых множеств (функция *maximum_independent_sets*), запускающий алгоритм библиотеки *igraph*, работающий немного быстрее обычного перебора. Также реализована функциональность подсчета числа независимости графа (функция *maximum_independent_set_size*), а также проверка заданного множества вершин в графе на независимое множество вершин наибольшего размера (функция *is_maximum_independent_set*).

Помимо этого, в данном файле реализована необходимая функциональность для решения задач вершинной раскраски графа, а именно, получение всех (если существуют) *k*-раскрасок графа (функция *k_coloring*) и проверка списка помеченных вершин на *k*-раскраску (функция *is_coloring*). При этом функция *k_coloring* дополнительно принимает необязательный параметр *break_on_single*, чтобы остановить поиск абсолютно всех *k*-раскрасок при нахождении первой из них. Для простоты дальнейшей реализации значение по умолчанию для данного параметра равно *True*.

Также данный файл имеет функциональность генерации случайных связных графов с определенным количеством вершин (функция *random_nx_graph*), а также наглядного просмотра определенного графа (функция *show_graph*). При этом функция *random_nx_graph* также принимает необязательный параметр *edge_randomizer* типа *Randomizer* из файла *utils/types/random.py* для настройки генератора случайных чисел для определения количества ребер графа: для различных задач данный генератор различен. Значение по умолчанию данного параметра является функция *uniform* из *utils/random.py*. Помимо этого, данная функция может принимать в качестве параметра не только определенное число вершин, а также пару чисел, обозначающих минимум и максимум количества вершин соответственно. При этом число вершин графа будет сгенерировано в данном отрезке случайно функцией *uniform* из *utils/random.py*.

Для создания уже готовых наборов данных, подающихся в нейронную сеть, создан файл *utils/graph_nn.py*. В данном файле реализована функция *generate_nx_graphs* генерации троек вида *(входной граф, выходной граф, граф без признаков)*. Входной граф содержит в себе признаки, необходимые для обучения. Так для решения задач о наибольшем независимом множестве и

вершинной раскраске вершинными признаками являются степени данных вершин, при этом реберные и глобальные признаки занулены из-за их избыточности. Выходной граф содержит в себе метки правильного ответа, также необходимого для обучения нейронной сети, а именно, подсчет точности и ошибки нейронной сети. Граф без признаков является начальным сгенерированным графом без меток и признаков, чаще всего нужный для визуализации сгенерированного графа. Также стоит отметить, что вершинные метки для решаемых задач закодированы *one-hot* кодировкой, которая будет описана ниже.

Еще одним нюансом является то, что при решении задачи о вершинной раскраске граф генерируется до тех пор, пока в нем не будет хотя бы одной *k*-раскраски. Т.к. задача о существовании *k*-раскраски решалась в [14], мы решаем задачу о генерации самой *k*-раскраски при условии ее существования.

3.2.2. Генерация случайных величин

Для определения функций, генерирующих случайные величины, необходимых функции *random_nx_graph*, из различных распределений, создан файл *utils/random.py*. В данном файле реализованы некоторые функции, генерирующие случайные величины из определенных распределений на ограниченном отрезке. Так, для генерации случайной величины количества вершин создана функция *uniform*, являющаяся генератором равномерно распределенной случайной величины на отрезке.

Помимо этого, также создана функция *wald*, возвращающая функцию, генерирующую случайную величину из пользовательски настроенного распределения, похожего на обратное нормальное распределение, но масштабированное на определенный отрезок и отзеркаленное относительно среднего значения отрезка. Описанное распределение случайной величины показано на следующем рисунке (см. рис. 3.1):

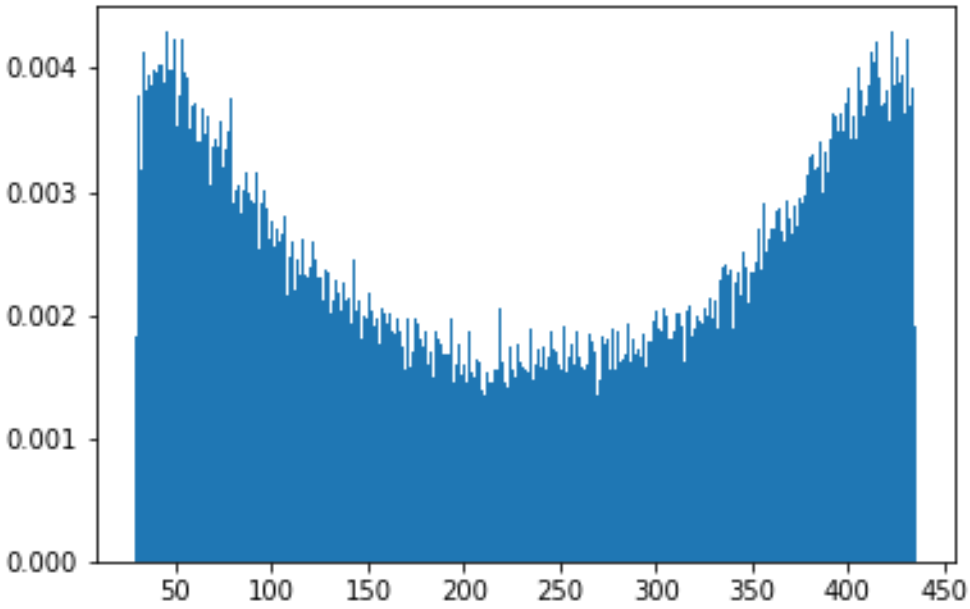

Рисунок 3.1 — Пользовательское распределение функции *wald*. По оси абсцисс находятся значения из отрезка [29, 435], по оси ординат — вероятность выпадения того или иного значения

Данное распределение необходимо для конфигурации числа ребер в сгенерированных графах в задаче нахождения наибольшего независимого множества для предоставления этих графов нейронной сети. Т.к. при добавлении или удалении ребра в графе с малым или большим количеством ребер ответ может измениться намного значительней, чем при той же самой операции в графе со средним количеством ребер, нейронная сеть, на вход которой попадают данные сгенерированные графы, не будет запоминать примерное количество ребер в ответе, что даст возможность получить действительно правдивый результат для графов любых размеров.

3.2.3. Остальная функциональность и константы

Для хранения дополнительных инструментов при работе с массивами создан файл *utils/numpy.py*. В данном файле реализован функционал для построения *one-hot* кодировки массива (функция *to_one_hot*) и создания массива фичей из списка необходимых полей и словаря данных для нейронной сети (функция *create_feature_array*). Функция *to_one_hot* при получении на вход массива a длины N выстраивает матрицу Y размера $N \times max(a) + 1$, для элементов которой верны следующие выражение:

$$Y_{i,j} = 1, j = a_i\ , \tag{3.1}$$

$$Y_{i,j} = 0, j \neq a_i. \tag{3.2}$$

Для хранения некоторого функционала создания и обслуживания нейронных сетей создан файл *utils/nn.py*. В данном файле реализован функционал корректировки скорости обучения (функция *create_lr_updater*). Данная функция необходима для уменьшения скорости обучения в определенное количество раз один раз в определенное количество эпох, чтобы нейронная сеть смогла обучаться и дальше. Помимо данной функции, также реализован функционал сохранения метрик обучения (функция *save_model_stats*), а также создания полносвязных нейронных сетей с нормализацией для GNN и GCN (функции *make_snt_mlp_model* и *make_keras_mlp_model*). Данные функции возвращают функции, создающие данные модели.

Для реализации функций общего назначения создан файл *utils/common.py*, в котором реализована единственная функция *enable_logger*, которая включает логирование работы Python-модуля в консоль. Эта функция удобна при дебаге и отслеживании процесса.

Также были созданы пользовательские типы данных, реализованные в папке *utils/types*. Файлы данной папки реализуют типы данных для конфигурации данных (*DataConfig*), обновления параметра скорости обучения (*LRConfig*), конфигурации процессов обучения моделей (*GraphNetsProcessConfig*, *DGLProcessConfig*), конфигурации полносвязных и сверточных слоев (*MLPConfig* и *ConvConfig* соответственно), а также тип функций генерации случайных чисел *Randomizer* и перечисляемый тип идентификации типа решаемой задачи (*SolutionFunction*).

Помимо этого, также реализован файл для хранения глобальных констант проекта *constants.py*. В данном файле хранятся константы названий ключей признаков и меток, необходимых в работе (*DEGREE_FEATURE*, *SOLUTION_FEATURE*, *FEATURES_KEY*), а также число классов *k* для задачи о вершинной *k*-раскраске (*K_COLORING_CLASSES*). Последняя константа установлена в данном файле для удобства доступа к ней из различных компонент проекта.

Далее перейдем к программной реализации экспериментов над моделями GNN.

3.3. Реализация компонент для запуска экспериментов над GNN

3.3.1. Принцип работы библиотеки graph_nets

Как было описано ранее, для реализации выбранных моделей GNN использовалась библиотека *graph_nets*, предоставляющая функционал для использования некоторых моделей, а также инструментарий построения собственных моделей на основе описанных в п. 2.1.1 блоков. Данные блоки бывают трех типов и предоставляют, помимо функций обновления состояний компонент, также функционал агрегации компонент. При построении блока существует возможность указать использование тех или иных компонент графа: флаги *use_edges*, *use_receiver_nodes*, *use_sender_nodes*, *use_globals* используются для добавления зависимостей блоков от реберных состояний, состояний входящих вершин, состояний исходящих вершин и глобального состояния соответственно. Данные состояния, как правило, конкатенируются при обработке отдельного графа, в результате чего модель обучается в зависимости вышеописанных состояний. После обработки массива состояний моделью каждый блок возвращает копию входного графа с измененными состояниями той или иной компоненты, за которую отвечает данный блок. После чего результат вызова данного блока направляется на вход блока другого типа в соответствии с моделью. Примером построения модели с помощью предоставленного блочного инструментария в данном проекте является сущность *GraphNetsMessagePassingNetwork* файла *models/graph_nets.py*, построенная согласно п. 2.1.4.

Помимо этого, данная библиотека работает с собственным типом данных *GraphsTuple*, представляющий собой набор независимых графов, объединенный в один мета-граф, хранящий характеристики компонент глобально для всех графов. Это позволяет работать с набором графов как с одним графом, что увеличивает производительность приложения.

3.3.2. Необходимые инструменты для реализации

Для хранения различных инструментов для работы с моделями *graph_nets* создан файл *utils/graph_nets.py*. В данном файле реализована функциональность для создания генератора наборов графов типа *GraphsTuple*, уже доступных для обработки GNN моделями (функция *graph_nets_data_gen*), вычисления различных метрик оценки работающей модели (функция *compute_accuracy*), подсчета функции ошибки (функция *create_loss_ops*), а также перевода объекта

типа *GraphsTuple* в объект данного типа с полями-тензорами, необходимыми для работы фреймворка обучения *Tensorflow* (функция *convert_to_tensor*). Стоит отметить, что функция *graph_nets_data_gen* в собственной реализации вызывает уже описанную функцию *generate_nx_graphs*, после чего производит конвертацию результатов в тип *GraphsTuple*, при этом, по аналогии с функцией *generate_nx_graphs*, в конфигурации при вызове принимает также распределение ребер. Также функция *create_loss_ops* высчитывает значение *softmax* функции, которая часто применяется в задачах классификации.

Для изучения точности модели при проведении экспериментов было построено несколько метрик. Первая метрика *correct* показывает, сколько правильных ответов в наборе выдала нейронная сеть, где правильным ответом считается любой ответ, удовлетворяющий условиям задачи и текущему графу. Например, в задаче о наибольшем независимом множестве правильным ответом считается любой набор множества вершин, являющийся независимым и размер которого равен размеру наибольшего независимого множества (по определению наибольшего независимого множества в п.1.1.1). Аналогично предыдущей задаче высчитывается по соответствующему определению для задачи о вершинной раскраске. Вторая метрика *solved* показывает, насколько ответ, выданный моделью, совпадает с ожидаемым ответом. Данная метрика, как правило, используется только для наблюдения за процессом обучения на начальных этапах, в которых реальная точность достаточно долгое время не изменяется. Данные метрики высчитываются в зависимости от решаемой задачи и возвращаются описанной функцией *compute_accuracy*.

Далее рассмотрим архитектуру приложения запуска экспериментов и ее компоненты.

3.3.3. Архитектура приложения запуска экспериментов над GNN

Данная архитектура представлена на рисунке ниже (см. рис. 3.2):

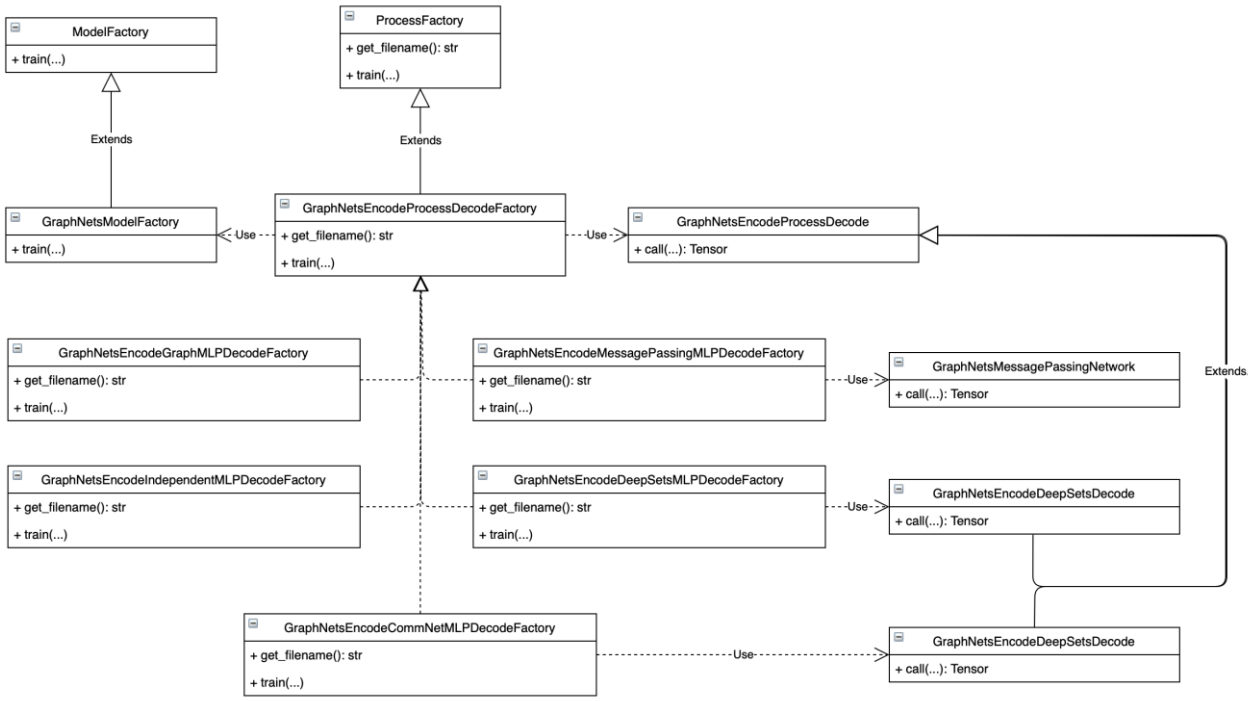

Рисунок 3.2 — Архитектура приложения запуска экспериментов над GNN

Для запуска экспериментов над выбранной моделью создается объект типа *ProcessFactory* соответственно модели. Данный абстрактный тип является прародителем любого типа запуска экспериментов любой модели в данном проекте и объявляет функции *get_filename* и *train* для получения файла с метриками процесса обучения и запуска процесса обучения модели соответственно. Прямым потомком данного типа является абстрактный тип *GraphNetsEncodeProcessDecodeFactory*, который создаёт и использует объекты типов *GraphNetsModelFactory* и *GraphNetsEncodeProcessDecode* для запуска процесса обучения и построения архитектуры моделей соответственно.

Тип *GraphNetsModelFactory* является наследником абстрактного типа *ModelFactory*, который является прародителем всех типов, выполняющих процесс обучения при определенных моделях и оптимизаторах, а также сохраняющих метрики обучения и логирование во время данного процесса. В частности, тип *GraphNetsModelFactory* реализует обучение моделей на основе моделей библиотеки *graph_nets*, а также сохранение метрик обучения в определенно поле данного типа.

Тип *GraphNetsEncodeProcessDecode* выстраивает архитектуру Encode-Decode, описанной в п.1.2.5, над исходной моделью. Прототип данной архитектуры показан на рисунке ниже (см. рис. 3.3):

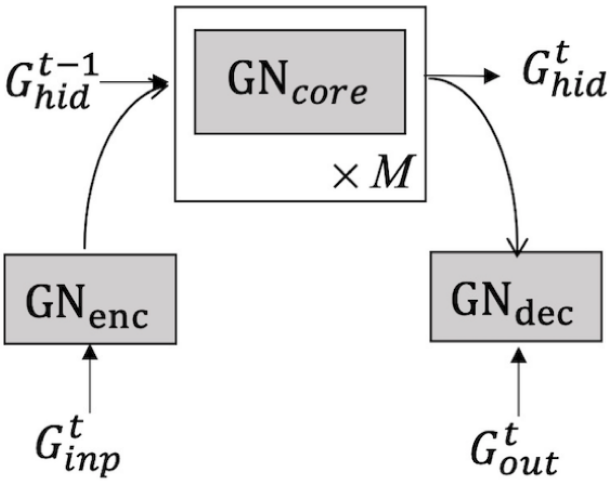

Рисунок 3.3 — Архитектура Encode-GNN-Decode. M — число слоев передачи сообщений

Как показано на данном рисунке, данная архитектура оборачивает модель Encoder-Decoder моделью. Помимо этого, при обработке графа результат энкодера конкатенируется с результатом каждого слоя передачи сообщений и подается на обработку следующему слою. Это дает возможность не потерять значения признаков компонент графа при обновлении состояний. Данную обертку реализует тип *GraphNetsEncodeProcessDecode*, который при создании объекта данного класса создает энкодер и декодер и при обработке графа выполняет вышеупомянутые шаги.

Для каждой модели GNN создается тип-наследник типа *ProcessFactory* (в данном случае из-за особенностей архитектуры типа *GraphNetsEncodeProcessDecodeFactory*) в соответствии с выбранной моделью. Например, для выбранной модели *GraphNetwork* реализован класс *GraphNetsEncodeGraphMLPDecodeFactory*, создающий нужное количество моделей-слоев одинакового типа для передачи сообщений и запускающий полное обучение и сохранение метрик обучения в определенный файл.

Также отдельно для типа *GraphNetsEncodeMessagePassingMLPDecodeFactory* реализована модель *GraphNetsMessagePassingNetwork*, реализующая алгоритм п.2.1.4. Помимо этого, также были реализованы дополнительные типы *GraphNetsDeepSetsEncodeProcessDecode* и *GrapheNetsCommNetEncodeProcessDecode*, являющиеся потомками типа *GraphNetsEncodeProcessDecode*, соответствующие одноименным моделям и исправляющие некоторые ошибки при использовании базовых моделей библиотеки *graph_nets*.

Для удобства конфигурирования эксперимента и его запуска создан файл *main-graph-nets.py*. В данном файле реализован функционал запуска процесса эксперимента над моделью с определенной конфигурацией — функция *start_process*. Данная функция принимает параметр в виде объекта типа *GraphNetsProcessConfig*, который позволяет конфигурировать следующие параметры:

- решаемую задачу (о наибольшем независимом множестве или вершинной раскраске);
- скорость обучения;
- конфигурацию данных: размер набора данных, максимальный и минимальный размеры генерируемых графов, распределение случайной величины количества ребер генерируемого графа;
- количество шагов передачи сообщений (*message-passing*);
- количество итераций обучения (эпох);
- папку сохранения файла с метриками обучения;
- паузу логирования и подсчета метрик во время обучения;
- тип используемого процесса (конкретный предок типа *GraphNetsEncodeProcessDecodeFactory*) и его настройку (количество и размер слоев полносвязной нейронной сети, описанной в п.1.2.1).

Это позволяет конфигурировать абсолютно весь процесс в одном месте. Пример данной конфигурации можно найти в функции *main* того же файла *main-graph-nets.py*, описанного в Приложении Б.

3.4. Реализация компонент для запуска экспериментов над GCN

3.4.1. Принцип работы библиотеки dgl

По аналогии с реализацией из предыдущего пункта, реализация компонент для запуска экспериментов над GCN также зависит от внешней библиотеки *dgl*. Данная библиотека предоставляет множество различных готовых моделей для их использования в проектах, а также собственный тип графовых данных *DGLGraph*. Данный тип данных схож по функциональности с *GraphsTuple* из предыдущего пункта и также позволяет сохранить набор графов в одном объекте для улучшения производительности обработки данных нейронной сетью. Помимо этого, данная библиотека предоставляет иной функционал для

построения механизма передачи сообщений вручную, а также поддерживает не только *Tensorflow*, но и *PyTorch* фреймворк.

3.4.2. Необходимые инструменты для реализации

По аналогии с идентичным п.3.3.2, для хранения дополнительного функционала, необходимого для работы, создан файл *utils/dgl.py*. В данном файле реализован идентичный функционал для создания генератора наборов графов типа *DGLGraph*, доступных для обработки GCN моделями (функция *dgl_data_gen*), вычисления различных метрик оценки работающей модели (функция *compute_accuracy*) и подсчета функции ошибки (функция *create_loss_ops*). Данные функции очень похожи по функционалу на идентичные по названию функции из п.3.3.2, однако данное разделение необходимо для поддержания различных типов данных, отвечающих за хранение и обработку графовых данных и являющихся уникальными для каждой библиотеки.

Также стоит отметить, что в данной интерпретации функции *dgl_data_gen* сгенерированные графовые данные типа *DGLGraph* не имеют реберных и глобальных признаков, что не противоречит выбранным моделям и задачам.

3.4.3. Архитектура приложения запуска экспериментов над GCN

Данная архитектура представлена на рисунке ниже (см. рис. 3.4):

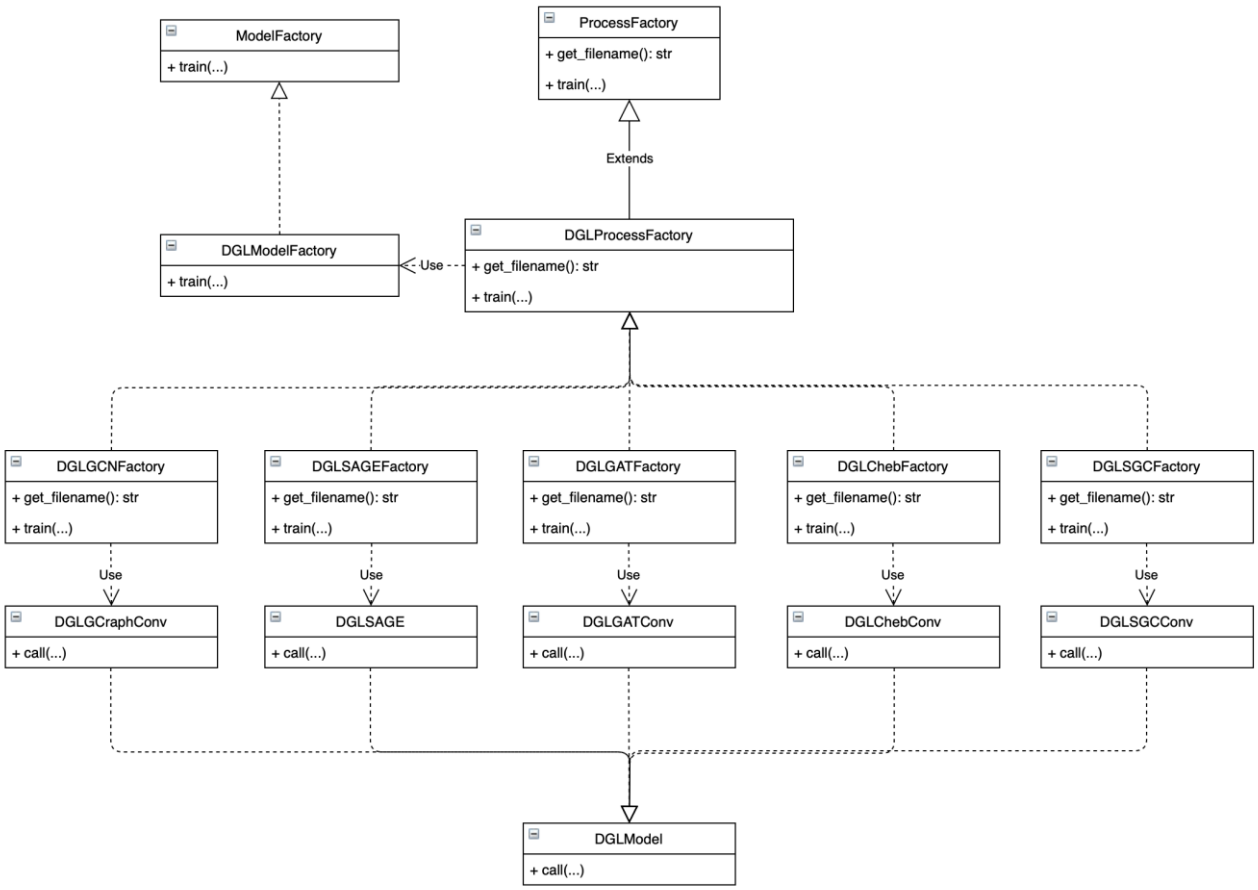

Рисунок 3.4 — Архитектура приложения запуска экспериментов над GCN

Аналогично архитектуре из п.3.3.3, прямым потомком типа *ProcessFactory*, являющегося прародителем любого типа запуска экспериментов, является абстрактный тип *DGLProcessFactory*, который создаёт и использует объект типа *DGLModelFactory* для запуска процесса обучения. Тип *DGLModelFactory*, также как и *GraphNetsModelFactory*, является наследником абстрактного типа *ModelFactory*, описанного ранее в п.3.3.3, и реализует обучение моделей на основе моделей библиотеки *dgl*.

Для каждой выбранной модели GCN создается тип-наследник типа *ProcessFactory* (в данном случае типа *DGLProcessFactory*) в соответствии с выбранной моделью. Помимо этого, также создается отдельный тип-обертка, являющийся прямым потомком абстрактного типа *DGLModel*, над выбранной моделью в соответствии с предыдущим типом. Данные типы-обертки необходимы для правильной реализации передачи сообщений и наслаивания моделей друг на друга. Например, при реализации запуска экспериментов над *GraphSAGE* создается тип *DGLSAGE* для реализации передачи сообщений путем наслаивания и последовательного выполнения однотипных моделей *SAGEConv* с различными параметрами и тип *DGLSAGEFactory* для запуска экспериментов, используя вышеописанный тип *DGLSAGE*.

Прототип данной архитектуры показан на рисунке ниже (см. рис. 3.5):

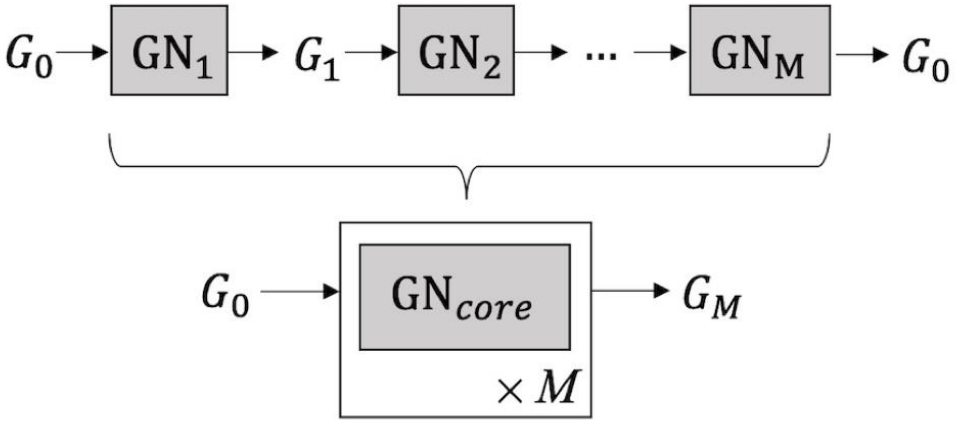

Рисунок 3.5 — Архитектура последовательных блоков. M — число слоев передачи сообщений

Как показано на данном рисунке, данная архитектура последовательно выполняет слои друг за другом, как это описано в определении в п.1.2.2. Для моделей GCN была выбрана именно эта архитектура из-за отсутствия поддержки библиотекой *dgl* фреймворка *Tensorflow* версий ниже 2, что позволило бы построить схожую архитектуру с использованием GCN.

Для удобства конфигурирования эксперимента и его запуска аналогично создан файл *main-dgl.py*. В данном файле реализован функционал запуска процесса эксперимента над моделью с определенной конфигурацией — функция *start_process*. По аналогии с реализацией запуска экспериментов с использованием библиотеки graph_nets, данная функция принимает параметр в виде объекта типа *DGLProcessConfig*, который позволяет конфигурировать те же параметры, которые были описаны в п.3.3.3, за исключением типа используемого процесса и его настройки: в данном случае тип процесса заменяется на конкретный предок типа *DGLProcessFactory*, а настройки позволяют конфигурировать не только размер скрытого сверточного слоя, но и размер входных данных (как правило, для текущих задач это 1) и функцию активации (по умолчанию, ReLU). Пример данной конфигурации можно найти в функции *main* того же файла *main-dgl.py*, описанного в Приложении Б.

ГЛАВА 4. ЗАПУСК И АНАЛИЗ РЕЗУЛЬТАТОВ ЭКСПЕРИМЕНТОВ

Как было описано ранее, конфигурация экспериментов с использованием модели определенного класса нейронных сетей (в данном случае, GNN или GCN) происходит в соответствующем данному классу main-файле: для класса GNN — *main-graph-nets.py*, для GCN — *main-dgl.py*.

Также стоит отметить, что перед запуском данных файлов необходимо установить переменную окружения *DGLBACKEND* в значение *tensorflow*. Это необходимо для установки среды выполнения обучения моделей *dgl*. Однако данное действие необходимо сделать также перед запуском экспериментов и над моделями GNN библиотеки *graph_nets* из-за общего подключения данных модулей внутри подключаемых модулей, что сделано для более удобного подключения.

4.1. Запуск и анализ экспериментов при решении задачи о наибольшем независимом множестве

Установлена следующая конфигурация обучения и тестирования:

- конфигурация данных:
  - размера набора данных — 128;
  - максимальный и минимальный размеры генерируемых графов — 20 и 30 соответственно;
  - распределение случайной величины количества ребер генерируемого графа — функция *wald*, распределение которой описано на рис. 3.1;
- конфигурация полносвязной нейронной сети для обновления состояний (для GNN):
  - количество полносвязных слоев — 2;
  - размер полносвязных слоев — 16;
- конфигурация сверточных слоев (для GCN):
  - размер входных данных — 1;
  - размер скрытого слоя — 16;
  - функция активации — ReLU;

- количество шагов передачи сообщений — конфигурируемое отдельно;
- количество итераций обучения — 1000;
- скорость обучения — 10^{-3}, уменьшается в 5 раз каждые 200 итераций для моделей GNN и 100 итераций для моделей GCN;
- пауза логирования и подсчета метрик во время обучения — 20 секунд.

Стоит отметить, что эксперименты проводились при различном количестве шагов передачи сообщений для наглядности пользы данного подхода в некоторых случаях. Эксперименты проводились при значениях количества шагов передачи сообщений равных 1, 5, 10 и 15.

Результаты данных экспериментов показаны в таблице ниже (см. табл. 1):

Таблица 1 — Значения точностей обученных моделей в задаче о наибольшем независимом множестве графа при различном количестве шагов передачи сообщений

Модели	Количество шагов передачи сообщений			
	1	5	10	15
GraphNetwork	1-2%	10-20%	20-30%	20-33%
GraphIndependent	0%	0%	0%	0%
MPNN	1-2%	12-20%	22-30%	28-33%
Deep Sets	0%	0%	0%	0%
CommNet	0-1%	15-20%	23-30%	33-40%
GraphConv	0%	0%	0%	0%
GraphSAGE	0%	1-2%	5-6%	10-15%
SGConv	0%	0%	0%	0%
ChebConv	0%	0%	0%	0%
GATConv	0%	0%	0%	0%

Первое, что стоит отметить в данной таблице, это прямая зависимость роста точности обучающихся моделей от количества шагов передачи сообщений. Это подтверждает работоспособность данной концепции в задачах на графах.

Второе, что также можно заметить в таблице, это обучаемость только тех моделей, которые связывают вершинные состояния в реляционном отношении и обновление которых зависит от собственных признаков. Так, например, не смогли обучиться модели Deep Sets и GraphIndependent из класса GNN из-за представления графа без связей в виде независимых вершин и их кодирования, что убирает всякие реляционные отношения между вершинами. Касательно моделей GCN, в данном случае смогла обучиться только модель GraphSAGE, т.к. в моделях GraphConv и GATConv новое состояние вершины не зависит от предыдущего, модель SGConv оказалась слишком простой для решения задачи, а модель ChebConv не смогла найти каких-либо зависимостей спектральных характеристик для решения задачи.

Третье, что можно заметить, это разница разброса точности результатов в моделях GraphNetwork, MPNN и CommNet. Это связано с занулением реберных и глобальных признаков графа, используемых в GraphNetwork и MPNN, что мешает обучиться модели лучше. Поэтому наименьший разброс точности среди данных моделей имеет CommNet, не использующая реберные и глобальные признаки.

График обучения модели CommNet выглядит следующим образом (см. рис. 3.6.):

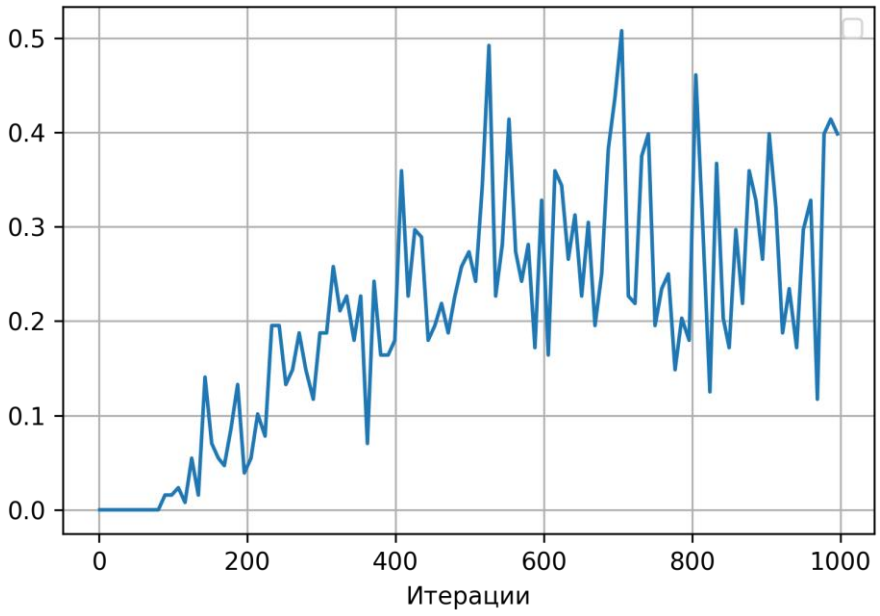

Рисунок 3.6 — График обучения обучения модели CommNet. По оси абсцисс — номер итерации обучения, по оси ординат — точность модели на данной итерации

Данный результат модели CommNet является достаточно неплохим результатом для задачи классификации множества вершин.

4.2. Запуск и анализ экспериментов при решении задачи о вершинной раскраске графа

Для запуска экспериментов над решением задачи о вершинной раскраске использовалась похожая конфигурация с некоторыми изменениями:

- конфигурация данных:
  - максимальный и минимальный размеры генерируемых графов — 10 и 15 соответственно;
  - распределение случайной величины количества ребер генерируемого графа — равномерное;
  - количество цветов раскраски — 3;
- конфигурация полносвязной нейронной сети для обновления состояний (для GNN):
  - количество полносвязных слоев — 4;
  - размер полносвязных слоев — 32;
- конфигурация сверточных слоев (для GCN):
  - размер скрытого слоя — 32.

Также к конфигурации количества шагов передачи сообщений добавилось значение 30.

В результате единственными моделями, у которых получилось немного обучиться на данной задаче, являются CommNet и MPNN с точностями 3-4% и 1-2% соответственно. Данные результаты могут быть связаны со сложностью задачи и слабостью нейронных сетей: необходимо использовать более объемные нейронные сети с большим количеством внутренних слоев большего размера, что также повлияет на время обучения.

ЗАКЛЮЧЕНИЕ

В данной научной работе была рассмотрена проблема решения задач, связанных с независимостью и вершинной раскраской в графах, и проведены эксперименты с различными нейронными сетями на данных задачах.

Для решения данных задач было решено использовать графовые нейронные сети и графовые сверточные нейронные сети из-за неевклидовости пространства графов и необходимости использовать реляционные связи между вершинами. В частности, было выбрано множество моделей из данных классов, а именно: GraphNetwork, GraphIndependent, MPNN, Deep Sets, CommNet из класса GNN и GraphConv, GraphSAGE, SGConv, ChebConv, GATConv из класса GCN.

Далее было реализовано приложение для запуска экспериментов с различной конфигурацией с использованием упомянутых моделей. После чего были проведены серии экспериментов с различной конфигурацией и проанализированы результаты данных экспериментов. В частности, было выявлено в задаче о наибольшем независимом множестве, что с использованием модели CommNet класса GNN удалось получить точность 30-40%, что является достаточно неплохим результатом и говорит о том, что нейронные сети могут быть применимы для решения такого типа задач и что указывает на расширение разных классов задач, к которым можно достаточно успешно применить нейронные сети.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Граф (математика) – Википедия // [Электронный ресурс]. – 2019/ Режим доступа: https://ru.wikipedia.org/wiki/Граф_(математика). – Дата доступа: 10.09.2019.
2. Задача о независимом множестве – Википедия // [Электронный ресурс]. – 2019/ Режим доступа: https://ru.wikipedia.org/wiki/Задача_о_независимом_множестве. – Дата доступа: 15.09.2019.
3. Раскраска графов // [Электронный ресурс]. – 2021/ Режим доступа: https://ru.wikipedia.org/wiki/Раскраска_графов. Дата доступа: 11.01.2021.
4. Искусственная нейронная сеть – Википедия // [Электронный ресурс]. – 2019/ Режим доступа: https://ru.wikipedia.org/wiki/Искусственная_нейронная_сеть. – Дата доступа: 11.10.2019.
5. Искусственный нейрон – Википедия // [Электронный ресурс]. – 2019/ Режим доступа: https://ru.wikipedia.org/wiki/Искусственный_нейрон. – Дата доступа: 12.10.2019.
6. Градиентный спуск – Википедия // [Электронный ресурс]. – 2019/ Режим доступа: https://ru.wikipedia.org/wiki/Градиентный_спуск. – Дата доступа: 12.10.2019.
7. Graph Neural Networks: A Review of Methods and Applications // [Электронный ресурс]. – 2019/ Режим доступа: https://arxiv.org/pdf/1812.08434.pdf. – Дата доступа: 18.03.2021.
8. How to do Deep Learning on Graphs with Graph Convolutional Networks // [Электронный ресурс]. – 2018/ Режим доступа: https://towardsdatascience.com/how-to-do-deep-learning-on-graphs-with-graph-convolutional-networks-7d2250723780. – Дата доступа: 25.10.2019.
9. Relational inductive biases, deep learning and graph networks // [Электронный ресурс]. – 2017/ Режим доступа: https://arxiv.org/pdf/1806.01261.pdf. – Дата доступа: 19.03.2021.
10. Neural Message Passing for Quantum Chemistry // [Электронный ресурс]. – 2018/ Режим доступа: https://arxiv.org/pdf/1704.01212.pdf. – Дата доступа: 21.03.2021.
11. NN Modules – PyTorch – DGL 0.6.0post1 documentation // [Электронный ресурс]. – 2019/ Режим доступа: https://docs.dgl.ai/api/python/nn.pytorch.html#module-dgl.nn.pytorch.conv. – Дата доступа: 25.03.2021.

12. Inductive Representation Learning on Large Graphs // [Электронный ресурс]. – 2018/ Режим доступа: https://arxiv.org/pdf/1706.02216.pdf. – Дата доступа: 25.03.2021.

13. Attention для чайников и реализация в Keras // [Электронный ресурс]. – 2019/ Режим доступа: https://habr.com/ru/post/458992/. – Дата доступа: 25.03.2021

14. Graph Coloring Meets Deep Learning: Effective Graph Neural Network Models for Combinatorial Problems // [Электронный ресурс]. – 2021/ Режим доступа: https://arxiv.org/pdf/1903.04598.pdf. – Дата доступа: 31.03.2021.

ПРИЛОЖЕНИЕ А

https://github.com/ElMaestro157/masters-degree

ПРИЛОЖЕНИЕ Б

Б.1. Исходный код файла main-graph-nets.py

```
from typing import Union, Type

import model_factories
from model_factories import GraphNetsEncodeGraphMLPDecodeFactory,
\
   GraphNetsEncodeMessagePassingMLPDecodeFactory, \
   GraphNetsEncodeIndependentMLPDecodeFactory,
GraphNetsEncodeDeepSetsMLPDecodeFactory, \
   GraphNetsEncodeCommNetMLPDecodeFactory

from constants import K_COLORING_CLASSES

import utils.random as random
from utils.graph_nets import graph_nets_data_gen
from utils.common import enable_logger

from utils.types.graph import SolutionFunction
from utils.types.nn import MLPConfig
from utils.types.config import LRConfig, DataConfig,
GraphNetsProcessConfig

GraphNetsProcessFactory = Type[Union[
   GraphNetsEncodeGraphMLPDecodeFactory,
   GraphNetsEncodeIndependentMLPDecodeFactory,
   GraphNetsEncodeMessagePassingMLPDecodeFactory,
   GraphNetsEncodeDeepSetsMLPDecodeFactory,
   GraphNetsEncodeCommNetMLPDecodeFactory
]]


def start_process(
       process_config: GraphNetsProcessConfig
):
   min_graph_size = process_config.data_config.min_graph
   max_graph_size = process_config.data_config.max_graph
   edge_randomizer = process_config.data_config.edge_randomizer
   data_gen = graph_nets_data_gen(
       (min_graph_size, max_graph_size),
       edge_randomizer=edge_randomizer,
   )
```

```
    solution_func = process_config.task

    lr = process_config.lr.value
    lr_step = process_config.lr.update_step
    lr_part = process_config.lr.update_part

    batch_size = process_config.data_config.batch
    epochs = process_config.epochs
    log_pause = process_config.log_pause

    process = process_config.process

    process_factory = process(
        encoder_config=MLPConfig(process_config.core.layer_size,
2),
        core_config=process_config.core,
        decoder_config=MLPConfig(process_config.core.layer_size,
2),
        lr_initial=lr,
        lr_update_step=lr_step,
        lr_update_part=lr_part,
        proc_steps=process_config.message_pass_steps,
        node_output_size=K_COLORING_CLASSES if solution_func ==
SolutionFunction.COLOR else 2,
        stats_dir=process_config.stats_dir
    )
    print(process_factory.get_filename())
    process_factory.train(
        data=data_gen,
        epochs=epochs,
        batch_size=batch_size,
        solution_func=solution_func,
        log_pause=log_pause
    )


def main():
    enable_logger(model_factories.__name__)

    scale = 130
    edge_randomizer = random.wald(scale)

    batch = 128
    min_graph_size = 20
    max_graph_size = 30
```

```
    lr = 1e-3

    proc_steps = [1, 5, 10, 15]
    configs = [
        GraphNetsProcessConfig(
            process=GraphNetsEncodeGraphMLPDecodeFactory,
            task=SolutionFunction.MAX_IND,
            core=MLPConfig(layer_size=16, layer_count=2),
            lr=LRConfig(
                value=lr,
                update_step=200,
                update_part=5
            ),
            data_config=DataConfig(
                batch=batch,
                min_graph=min_graph_size,
                max_graph=max_graph_size,
                edge_randomizer=edge_randomizer
            ),
            message_pass_steps=proc_step,
            epochs=1000,
            log_pause=20,
            stats_dir='./data/graph-nets/max-ind'
        )
        for proc_step in proc_steps
    ]

    for config in configs:
        start_process(config)


if __name__ == '__main__':
    main()
```

Б.2. Исходный код файла main-dgl.py

```
from typing import cast, Union, Type

import tensorflow as tf

import model_factories

from utils.dgl import dgl_data_gen
```

```
import utils.random as random

from constants import K_COLORING_CLASSES

from utils.common import enable_logger

from utils.types.graph import SolutionFunction
from utils.types.nn import ConvConfig
from utils.types.config import LRConfig, DataConfig,
DGLProcessConfig

DGLProcessFactory = Type[Union[
   model_factories.DGLGCNFactory,
   model_factories.DGLGATFactory,
   model_factories.DGLSAGEFactory,
   model_factories.DGLChebFactory,
   model_factories.DGLSGCFactory
]]


def start_process(
       process_config: DGLProcessConfig
):
   min_graph_size = process_config.data_config.min_graph
   max_graph_size = process_config.data_config.max_graph
   edge_randomizer = process_config.data_config.edge_randomizer
   data_gen = dgl_data_gen(
       (min_graph_size, max_graph_size),
       edge_randomizer=edge_randomizer,
       norm=False
   )
   solution_func = process_config.task

   lr = process_config.lr.value
   lr_step = process_config.lr.update_step
   lr_part = process_config.lr.update_part

   batch_size = process_config.data_config.batch
   epochs = process_config.epochs
   log_pause = process_config.log_pause

   process = process_config.process

   process_factory = process(
       core_config=process_config.core,
```

```
        lr_initial=lr,
        lr_update_step=lr_step,
        lr_update_part=lr_part,
        proc_steps=process_config.message_pass_steps,
        node_output_size=K_COLORING_CLASSES if solution_func ==
SolutionFunction.COLOR else 2,
        stats_dir=process_config.stats_dir
    )
    print(process_factory.get_filename())
    process_factory.train(
        data=data_gen,
        epochs=epochs,
        batch_size=batch_size,
        solution_func=solution_func,
        log_pause=log_pause
    )


def main():
    enable_logger(model_factories.__name__)

    scale = 130
    edge_randomizer = random.wald(scale)

    batch = 128
    min_graph_size = 20
    max_graph_size = 30

    lr = 1e-3

    proc_steps = [1, 5, 10, 15]
    configs = [
        DGLProcessConfig(
            process=model_factories.DGLSAGEFactory,
            task=SolutionFunction.MAX_IND,
            core=ConvConfig(
                input_layer_size=1,
                hidden_layer_size=16,
                activation=tf.nn.relu
            ),
            lr=LRConfig(
                value=lr,
                update_step=100,
                update_part=5
            ),
```

```
            data_config=DataConfig(
                batch=batch,
                min_graph=min_graph_size,
                max_graph=max_graph_size,
                edge_randomizer=edge_randomizer
            ),
            message_pass_steps=proc_step,
            epochs=1000,
            log_pause=20,
            stats_dir='./data/dgl/max-ind'
        )
        for proc_step in proc_steps
    ]

    for config in configs:
        start_process(config)


if __name__ == '__main__':
    main()
```