Санкт-Петербургский государственный университет

Прикладная математика, программирование и искусственный интеллект

Отчет по учебной практике 2 (научно-исследовательской работе) (семестр 2)

Тема:
Графовые нейронные сети

Выполнил:
Громов Григорий Андреевич, группа 22.Б05

Научный руководитель:
Мокаев Тимур Назирович

Кафедра прикладной кибернетики

Санкт-Петербург

2023

Введение

Передо мной была поставлена задача углубиться в изучение нейронных сетей и разобраться в теме графовых нейронных сетей.

Постановка задачи:

- Изучить графовые нейронные сети (GNN): зачем они нужны и в каких задачах используются
- Научиться реализовывать GNN на Python
- Решить какую-то из задач (например, задачу классификации) с помощью GNN

Основная часть:

В связи с тем, что моя прошлая НИР была на похожую тему, и в ней я детально разбирался с механизмами работы базовой нейронной сети, позволю себе перейти сразу к ее усложненной версии – графовой нейронной сети.

Но перед этим необходимо разобраться с тем, что мы будем подразумевать под словом «граф»:

Граф – это тройка, состоящая из двух множеств (множества вершин и множество ребер) и правила, сопоставляющего каждому ребру пару вершин.

Говоря проще, граф – особенный способ представления данных и взаимосвязей между ними:

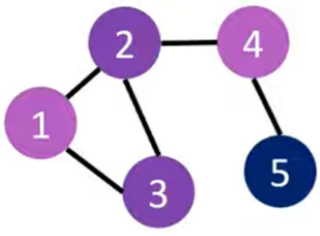


рис. 1

Неориентированные графы мы будем хранить с помощью *матрицы связности*:

	V1	V2	...
V1	0	1	...
V2	1	0	...
V3	1	1	...
...	...	...	...

рис. 2

Если на пересечении столбцов i-ой и j-ой вершин стоит 1 – значит они связаны, 0 – значит нет.

Вершины и ребра могут иметь дополнительные свойства, называемые *признаками вершин* и *признаками ребер* соответственно. Например, если граф представляет молекулу (атомы + связи между ними), то признаками его вершин могут являться тип атома, количество протонов и т.д. Признаками же ребер могут являться свойства связей – ковалентная или полярная, прочность и т.д.

Графы используются для представления и обработки данных в:

1. Медицине и фармацевтике

2. Компаниях, занимающихся продажами в интернете (Avito, Яндекс Маркет, Ozon) для составления системы рекомендаций

3. Социальных сетях: выдача новостей интересных вам, предложения относительно возможных друзей

4. Современных 3D играх (GTA5) и ПО для 3D-моделирования (Blender)

Что могут делать графовые нейронные сети:

1. Предсказание свойств вершины

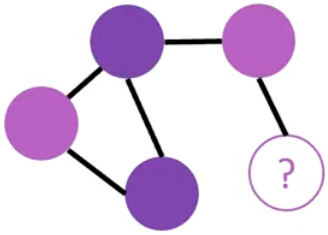


рис. 3

(Например, курит ли человек N?)

2. Предсказание связей

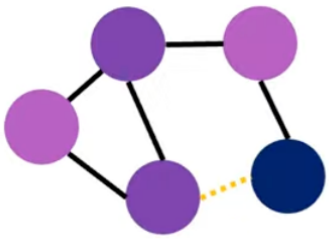


рис. 4

(Обычно связь между клиентом и товаром, например, какой товар предложить клиенту для покупки)

3. Классификация графов

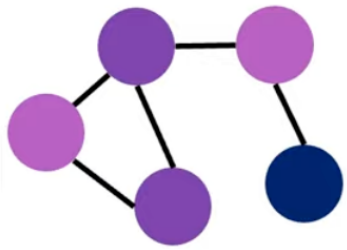


рис. 5

(Часто используется в медицине для прогнозирования свойств соединений)

Главная идея GNN

Используя всю информацию о графе (в том числе признаки вершин и ребер) GNN выводит новое представление (называемое вложением) для каждой вершины. Для каждой вершины мы производим сбор информации о признаках ее соседей (вершин, соединенных с данной ребром) и специальным образом объединяем ее, получая новые признаки данной вершины – вложение. Такой процесс называется *сверткой графа*.

Рассмотрим простой пример:

Предположим, что начальный граф состоит из одной желтой, одной зеленой и трех синих вершин:

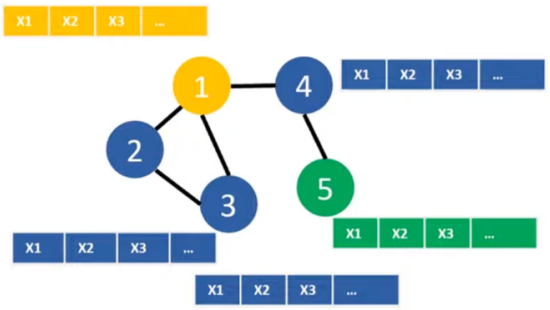


рис. 6

Цветные прямоугольники с секциями около вершин – признаки вершин (x1 – первый признак, x2 – второй признак и т.д.)

Сосредоточимся на желтой вершине и обновим ее состояние. Для этого необходимо собрать информацию о ее соседях и определенным образом сопоставить ее с исходным состоянием.

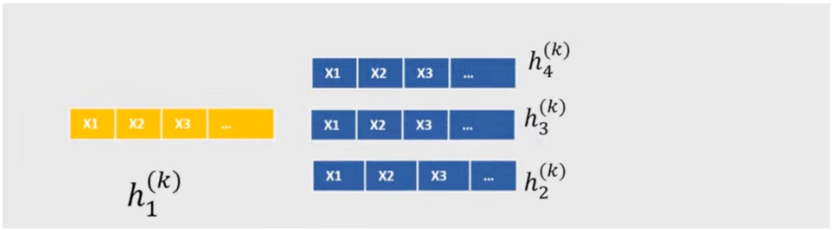


рис. 7

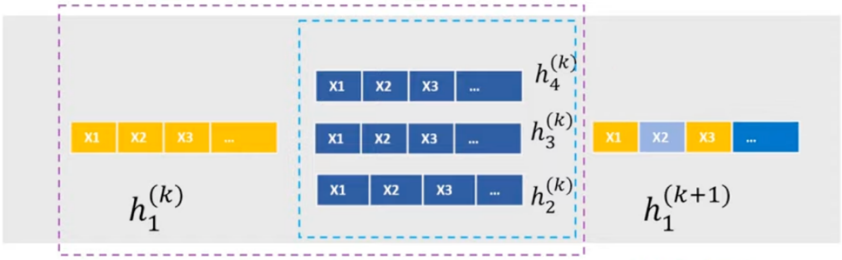


рис. 8

h1(k+1) – новое состояние желтой вершины.

Произведя аналогичные действия с остальными вершинами, мы получим новый граф:

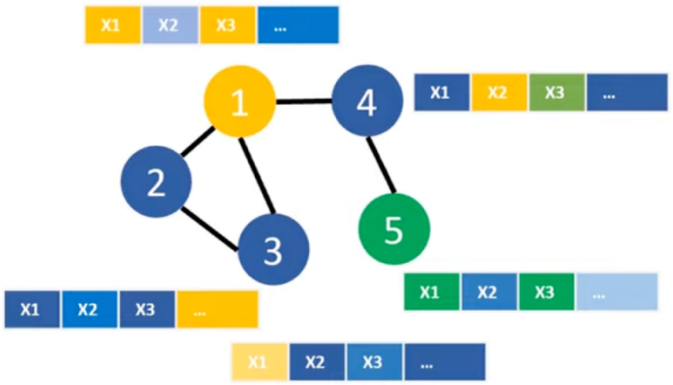


рис. 9

Обратите внимание, что, так как была произведена лишь одна итерация, информацию о зеленой вершине знает пока лишь вершина 4, так как она единственный прямой сосед зеленой вершины (вершины 5).

Двигаемся дальше:

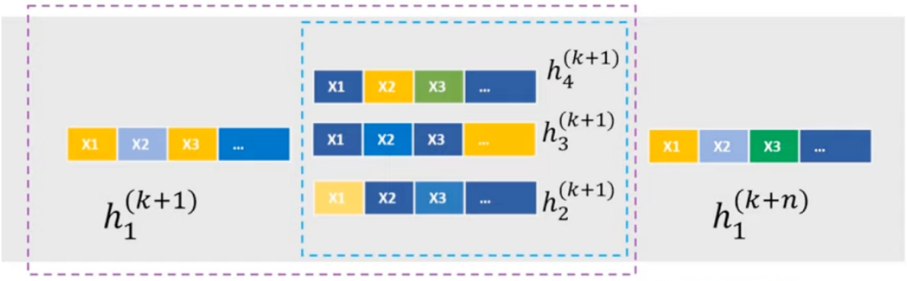


рис. 10

Так как после первой итерации прямой сосед желтой вершины (вершина 4) уже что-то знала о зеленой вершине, то после второй итерации сама желтая вершина узнает какую-то информацию о зеленой.

Однако при таком подходе важно грамотно подбирать количество повторений таких итераций, потому что при их чрезмерном количестве все вершины как бы «сравняются» и будут иметь одинаковые признаки.

Если формулировать более точно, то для обновления признаков вершины у нас должно быть 2 операции:

Aggregate – собирает информацию о соседях на данном уровне
Update – обновляет признаки вершины в зависимости от текущих признаков вершины и признаков вершины, обработанных с помощью операции Aggregate:

$$h_u^{(k+1)} = UPDATE^{(k)}\left(h_u^{(k)}, AGGREGATE^{(k)}(\{h_v^{(k)}, \forall\, v \in \mathcal{N}(u)\})\right)$$

$h_u^{(k+1)}$ – признаки вершины u на временном шаге k+1

Это означает, что после нескольких таких итераций, каждая вершина будет знать что-то о других вершинах. И полученных в результате таких преобразований граф уже можно будет использовать для прогнозирования.

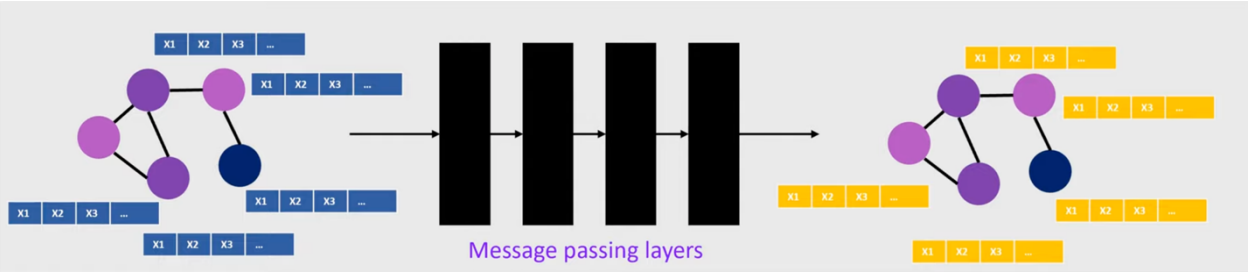


рис. 11

Создание GNN на Python

Данные

В этой работе мы будем создавать нейронную, которая будет обрабатывать данные о молекулах и предсказывать их коэффициент растворимости в воде.

Мы будем использовать PyTorch для создания нейронной сети, а также RDKit для обработки данных о молекулах. Код будем писать в среде разработки Google Collab.

Набор данных о молекулах для обучения и тестирования нейронной сети возьмем из коллекции PyTorch Geometric под названием ESOL. Это набор данных о растворимости в воде 1128 соединений.

Импортируем этот набор и запишем его в массив data:

```
import rdkit
from torch_geometric.datasets import MoleculeNet

data = MoleculeNet(root=".", name="ESOL")
```

Информация о соединении представляется в следующем виде:

Для наглядности рассмотрим на пример соединения имеющего номер 81:

- Информация о признаках вершин:

```
data[81].x
```

```
tensor([[6, 0, 4, 5, 3, 0, 4, 0, 0],
        [6, 0, 3, 5, 0, 0, 3, 1, 1],
        [7, 0, 2, 5, 0, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [7, 0, 2, 5, 0, 0, 3, 1, 1],
        [6, 0, 3, 5, 0, 0, 3, 1, 1],
        [7, 0, 2, 5, 0, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [6, 0, 3, 5, 1, 0, 3, 1, 1],
        [7, 0, 2, 5, 0, 0, 3, 1, 1],
        [6, 0, 3, 5, 0, 0, 3, 1, 1]])
```

- Информация о связях:

```
data[81].edge_index.t()
```

```
tensor([[ 0,  1],
        [ 1,  0],
        [ 1,  2],
        [ 1, 10],
        [ 2,  1],
        [ 2,  3],
        [ 3,  2],
        [ 3,  4],
        [ 4,  3],
        [ 4,  5],
        [ 5,  4],
        [ 5,  6],
        [ 5, 10],
        [ 6,  5],
        [ 6,  7],
        [ 7,  6],
        [ 7,  8],
        [ 8,  7],
        [ 8,  9],
        [ 9,  8],
        [ 9, 10],
        [10,  1],
        [10,  5],
        [10,  9]])
```

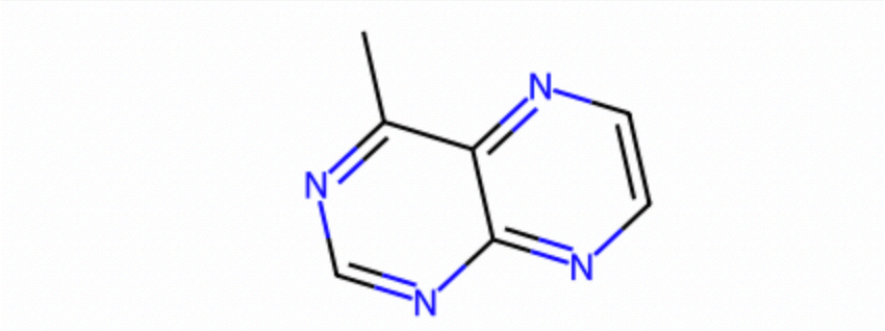
- Информация о растворимости:

```
data[81].y
```

```
tensor([[-0.4660]])
```

Помимо такого формального представления мы можем вывести структурную формулу данного соединения с помощью пакета RDKit (он преобразует информацию о молекуле из формата SMILES в привычную схему):

```
from rdkit import Chem
from rdkit.Chem.Draw import IPythonConsole
molecule = Chem.MolFromSmiles(data[81]["smiles"])
molecule
```



Реализация

Наша нейронная сеть будет оценивать растворимость соединения в воде – то есть проводить классификацию графа. Поэтому нам нужно объединить все признаки графа в одну структуру данных. Важно выбрать такой механизма, чтобы он мог использоваться для графа с любым количеством вершин. Предлагается следующий механизм:

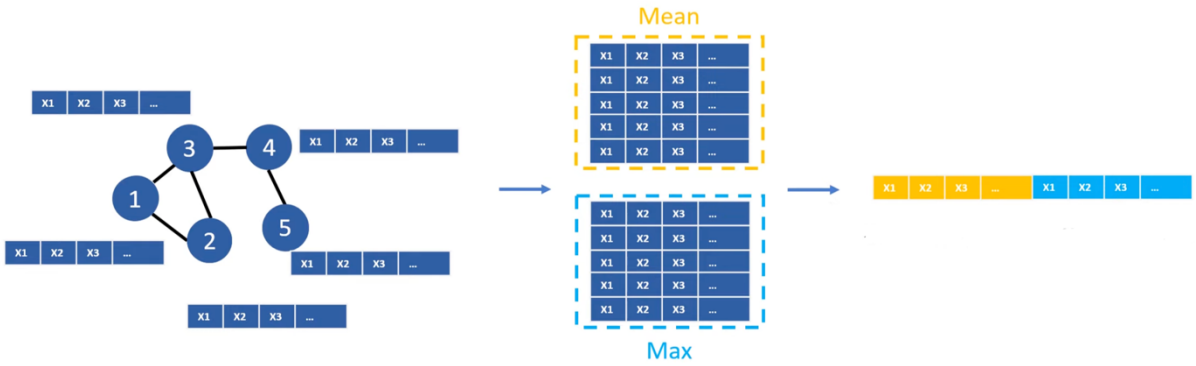


рис. 12

Мы вычисляем среднее и максимальное значения по каждой характеристике признака (в нашем случае у каждой вершины признак состоит из девяти характеристик) и последовательно объединяем их в один вектор. Таким образом результирующий вектор будет в два раза длинней исходного.

Для начала подключим все необходимые модули и библиотеки:

```
import torch
import torch.nn.functional as F

from torch.nn import Linear
from torch_geometric.nn import GCNConv, TopKPooling, global_mean_pool
from torch_geometric.nn import global_mean_pool as gap, global_max_pool as gmp
from torch_geometric.datasets import MoleculeNet
```

GCNConv – это простой сверточный слой графа

Далее определим две стандартные для PyTorch функции – init (определение слоев нейронной сети) и forward (функция для обмена информацией между сломи):

```
def __init__(self):

    super(GCN, self).__init__()
    torch.manual_seed(42)

    #определяем слои передачи информации:

    #слой преобразования (он преобразует признаки вершин (9 характеристик)
    #в "вектор внедрения" длиной 64)
    self.initial_conv = GCNConv(data.num_features, embedding_size)

    #остальные слои передачи инорамации:
    self.conv1 = GCNConv(embedding_size, embedding_size)
    self.conv2 = GCNConv(embedding_size, embedding_size)
    self.conv3 = GCNConv(embedding_size, embedding_size)

    # выходной слой:
    self.out = Linear(embedding_size*2, 1)
```

В последней строчке присутствует умножение длины вектора внедрения на 2, потому что, как мы обсуждали ранее, размер выходного вектора будет в 2 раза больше.

```
def forward(self, x, edge_index, batch_index):

    #передача инормации в первый слой:
    update_vertex_state = self.initial_conv(x, edge_index)
    update_vertex_state = F.tanh(update_vertex_state) #функция активации

    #передача информации в остальные слои:
    update_vertex_state = self.conv1(update_vertex_state, edge_index)
    update_vertex_state = F.tanh(update_vertex_state)
    update_vertex_state = self.conv2(update_vertex_state, edge_index)
    update_vertex_state = F.tanh(update_vertex_state)
    update_vertex_state = self.conv3(update_vertex_state, edge_index)
    update_vertex_state = F.tanh(update_vertex_state)

    # объединим два набора характеристик вершин в один вектор:
    update_vertex_state = torch.cat([gmp(update_vertex_state, batch_index), # максимальный
                        gap(update_vertex_state, batch_index)], dim=1) # средний

    # выходной слой:
    out = self.out(update_vertex_state)

    return out, update_vertex_state
```

x - признаки вершины
edge_index - информация о ребрах
batch_index - индекс группы, необходимый для выбора вершин, подходящий для объединения

F.tanh – функция активации

Обе функции для удобства помести в класс GCN(torch.nn.Module)

Теперь нам необходимо обучить нашу нейронную сеть

Подготовим окружение:

```
model = GCN()

from torch_geometric.data import DataLoader
import warnings
warnings.filterwarnings("ignore")

# Определяем функцию, вычисляющую ошибку:
loss_fn = torch.nn.MSELoss()
# Определяем функцию обновления на основе ошибки и указываем скорость обучения:
optimizer = torch.optim.Adam(model.parameters(), lr=0.0007)

# Используем графический процессор, если он доступен:
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Определим, каким образом мы будем передавать данные для обучения нейронной сети:
data_size = len(data)

#  Размер одной партии, которую мы будем передавать
NUM_GRAPHS_PER_BATCH = 64 # то есть будем передавать по 64 молекулы

#  80% пойдут на обучение:
loader = DataLoader(data[:int(data_size * 0.8)],
                    batch_size=NUM_GRAPHS_PER_BATCH, shuffle=True)
# оставшиеся 20% для тестирования:
test_loader = DataLoader(data[int(data_size * 0.8):],
                         batch_size=NUM_GRAPHS_PER_BATCH, shuffle=True)
```

Сама функция:

```
def train(data):

    for batch in loader:
      #Будем использовать графический процессор:
      batch.to(device)
      #Сбрасываем градиент:
      optimizer.zero_grad()
      #Обновляем вектора внедрения:
      pred, embedding = model(batch.x.float(), batch.edge_index, batch.batch)
      #Вычисляем ошибку:
      loss = loss_fn(pred, batch.y)
      loss.backward()
      #На основе градиента обновляем модель:
      optimizer.step()
    return loss, embedding
```

Запусти процесс обучения:

```
print("Starting training...")
losses = []
for epoch in range(2000):
    loss, h = train(data)
    losses.append(loss)
    if epoch % 100 == 0:
      print(f"Epoch {epoch} | Train Loss {loss}")
```

Будем выводить размер ошибки для каждого сотого круга:

```
Starting training...
Epoch 0 | Train Loss 11.665949821472168
Epoch 100 | Train Loss 0.8536352515220642
Epoch 200 | Train Loss 1.118796467781067
Epoch 300 | Train Loss 0.32378262281417847
Epoch 400 | Train Loss 0.23642905056476593
Epoch 500 | Train Loss 0.3853507936000824
Epoch 600 | Train Loss 0.2631867229938507
Epoch 700 | Train Loss 0.07353740930557251
Epoch 800 | Train Loss 0.13420253992080688
Epoch 900 | Train Loss 0.14671717584133148
Epoch 1000 | Train Loss 0.040488649159669876
Epoch 1100 | Train Loss 0.04992757737636566
Epoch 1200 | Train Loss 0.15233300626277924
Epoch 1300 | Train Loss 0.04923718050122261
Epoch 1400 | Train Loss 0.009802093729376793
Epoch 1500 | Train Loss 0.034630656242370605
Epoch 1600 | Train Loss 0.04991237446665764
Epoch 1700 | Train Loss 0.012065283954143524
Epoch 1800 | Train Loss 0.028051361441612244
Epoch 1900 | Train Loss 0.072029247879982
```

Видно, что по ходу обучение размер ошибки уменьшается

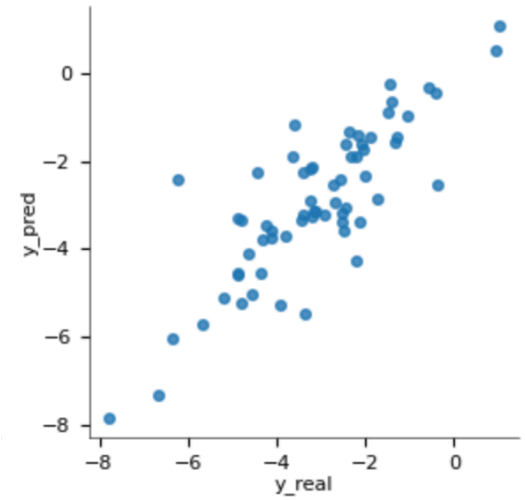
Протестируем обученную нейронную сеть:

```
import pandas as pd

test_batch = next(iter(test_loader))
with torch.no_grad():
    test_batch.to(device)
    pred, embed = model(test_batch.x.float(), test_batch.edge_index, test_batch.batch)
    df = pd.DataFrame()
    df["y_real"] = test_batch.y.tolist()
    df["y_pred"] = pred.tolist()
df["y_real"] = df["y_real"].apply(lambda row: row[0])
df["y_pred"] = df["y_pred"].apply(lambda row: row[0])
df
```

index	y_real	y_pred
0	0.9399999976158142	0.4927942454814911
1	-6.236999988555908	-2.419463872909546
2	-2.5399999618530273	-3.1966168880462646
3	-5.679999828338623	-5.727325439453125
4	-2.4600000381469727	-1.622434139251709
5	-4.882999897003174	-4.572348117828369
6	-4.798999786376953	-3.3411951065063477
7	-2.490000009536743	-3.598332166671753
8	-2.2200000286102295	-1.8897793292999268
9	-4.800000190734863	-5.245016574859619
10	-2.3399999141693115	-1.8775300979614258
11	-7.800000190734863	-7.8357625007629395
12	-0.5899999737739563	-0.3153363764286041
13	-2.380000114440918	-1.3514679670333862
14	-3.7899999618530273	-3.6862335205078125
15	-4.119999885559082	-3.57407808303833
16	-4.098999977111816	-3.7527573108673096
17	-3.3499999046325684	-5.466592788696289
18	-4.880000114440918	-3.301300525665283
19	-2.9200000762939453	-3.2214932441711426
20	-0.4000000059604645	-0.45377346873283386
21	-4.230000019073486	-3.477742910385132
22	1.0199999809265137	1.0705498456954956
23	-3.239000082015991	-2.898294448852539
24	-4.570000171661377	-5.038523197174072

Мы видим, что в подавляющем большинстве случаев наша показания нашей модели очень близки к истине. Это также можно увидеть на графике. В идеале должна быть прямая под углом в 45 градусов, но и в наших результат вполне прослеживается похожая зависимость:



Список источников:

https://neerc.ifmo.ru/wiki/index.php?title=Графовые_нейронные_сети
https://habr.com/ru/companies/vk/articles/557280/
https://academy.yandex.ru/handbook/ml/article/grafovye-nejronnye-seti
https://www.youtube.com/watch?v=ERvUf_lNopo
https://youtu.be/fOctJB4kVlM?si=y-FtVd2mywTwY29f
https://www.youtube.com/watch?v=ABCGCf8cJOE&list=PLV8yxwGOxvvoNkzPfCx2i8an--Tkt7O8Z&index=2
https://www.youtube.com/watch?v=YdGN-J322y4
https://www.youtube.com/watch?v=0YLZXjMHA-8&list=PLV8yxwGOxvvoNkzPfCx2i8an--Tkt7O8Z&index=3

Картинки рис.1-рис.12 взяты из нескольких видео:
https://www.youtube.com/watch?v=fOctJB4kV_lM
https://www.youtube.com/watch?v=ABCGCf8cJOE
https://www.youtube.com/watch?v=0YLZXjMHA-8