УДК 519.68:007.5

Лабораторная работа 6. Графовые нейронные сети

Цели работы: знакомство с графовыми нейронными сетями и типовыми задачи их приложения, а также сравнение производительности и качества работы графовых сетей с неспециализированными нейросетевыми моделями.

Ключевые слова: сверточные графовые слои, классификация на графах, молекулярные графы, трансформеры на графах.

§ 1. Классификация вершин графов с помощью графовых сетей

1.1. Подготовка к работе. Введем следующие команды в рабочей тетради jupyter и проверим, что они исполнились без ошибок:

```
import os
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Как вариант, можно вместо локально установленного Python'а использовать облачный сервис Google Colab с его реализацией тетради jupyter.

```
zip_file = keras.utils.get_file(
    fname="cora.tgz",
    origin="https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz",
    extract=True,
)
data_dir = os.path.join(os.path.dirname(zip_file), "cora_extracted/cora")
print(data_dir)
```

Загрузим данные по цитированию в формате Pandas фреймов.

```
citations = pd.read_csv(
    os.path.join(data_dir, "cora.cites"),
    sep="\t",
    header=None,
    names=["target", "source"],
)
print("Citations shape:", citations.shape)
```

Распечатаем фрагмент датафрейма с цитатами. Заметим, что колонках source отмечены идентефикаторы статей источников ссылок, а в колонках target — статей на которые они ссылаются.

```
citations.sample(frac=1).head()
```

Загрузим данные по статьям и их темам в формате Pandas фреймов.

```
column_names = ["paper_id"] + [f"term_{idx}" for idx in range(1433)] +
                ["subject"]
papers = pd.read_csv(
    os.path.join(data_dir, "cora.content"), sep="\t", header=None,
                    names=column_names,
)
print("Papers shape:", papers.shape)
```

Распечатаем фрагмент датафрейма по статьям. В ряду subject перечислены все возможные темы, которые могут встречаться в статьях, а в колонках term отмечена встречаемость (0 или 1) соотвествующих слов из словаря.

```
print(papers.sample(5).T)
```

Выведем количество статей, в которых представлены соответвующие темы:

```
print(papers.subject.value_counts())
```

Преобразуем систему индексации статей и ссылок к стартующей с нуля:

```
class_values = sorted(papers["subject"].unique())
class_idx = {name: id for id, name in enumerate(class_values)}
paper_idx = {name: idx for idx, name in
        enumerate(sorted(papers["paper_id"].unique()))}

papers["paper_id"] = papers["paper_id"].apply(lambda name: paper_idx[name])
citations["source"] = citations["source"].apply(lambda name: paper_idx[name])
citations["target"] = citations["target"].apply(lambda name: paper_idx[name])
papers["subject"] = papers["subject"].apply(lambda value: class_idx[value])
```

Визуализируем граф цитирований для небольшого фрагмента общего графа. Цветом будем отображать тематику статей.

```
plt.figure(figsize=(10, 10))
colors = papers["subject"].tolist()
cora_graph = nx.from_pandas_edgelist(citations.sample(n=1500))
subjects = list(papers[papers["paper_id"].isin(list(cora_graph.nodes))]
                        ["subject"])
nx.draw_spring(cora_graph, node_size=15, node_color=subjects)
```

Подготовим датасеты для обучения и проверки моделей.

```
train_data, test_data = [], []

for _, group_data in papers.groupby("subject"):
    # Выберем 50% всех примеров для обучения.
    random_selection = np.random.rand(len(group_data.index)) <= 0.5
    train_data.append(group_data[random_selection])
    test_data.append(group_data[~random_selection])

train_data = pd.concat(train_data).sample(frac=1)
test_data = pd.concat(test_data).sample(frac=1)
print("Train data shape:", train_data.shape)
print("Test data shape:", test_data.shape)
```

1.2. Подготовка к обучению. Зададим общие параметры для моделей.

```
hidden_units = [32, 32]
```

```
learning_rate = 0.01
dropout_rate = 0.5
num_epochs = 300
batch_size = 256
```

Реализуем функцию для обучения произвольной модели на наших данных.

```
def run_experiment(model, x_train, y_train):
    # Будем использовать метод Adam во всех моделях для начала
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate),
        loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")],
    )
    # Предусмотрим раннюю остановку в моделях.
    early_stopping = keras.callbacks.EarlyStopping(
        monitor="val_acc", patience=50, restore_best_weights=True
    )
    # Fit the model.
    history = model.fit(
        x=x_train,
        y=y_train,
        epochs=num_epochs,
        batch_size=batch_size,
        validation_split=0.15, # выделение на валидацию 15% примеров
        callbacks=[early_stopping],
    )

    return history
```

Также предусмотрим функцию для визуализации процесса обучения модели.

```
def display_learning_curves(history):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    ax1.plot(history.history["loss"])
    ax1.plot(history.history["val_loss"])
    ax1.legend(["train", "test"], loc="upper right")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")

    ax2.plot(history.history["acc"])
    ax2.plot(history.history["val_acc"])
    ax2.legend(["train", "test"], loc="upper right")
    ax2.set_xlabel("Epochs")
    ax2.set_ylabel("Accuracy")
    plt.show()
```

1.3. Референсная полносвязная модель для классификации вершин графа. Реализуем универсальную функцию для создания полносвязного персептронного слоя с нормализацией и оциональным дропаутом.

```
def create_ffn(hidden_units, dropout_rate, name=None):
    fnn_layers = []

    for units in hidden_units:
        fnn_layers.append(layers.BatchNormalization())
        fnn_layers.append(layers.Dropout(dropout_rate))
        fnn_layers.append(layers.Dense(units, activation=tf.nn.gelu))

    return keras.Sequential(fnn_layers, name=name)
```

Подготовим данные для обучения референсной модели.

```
feature_names = list(set(papers.columns) - {"paper_id", "subject"})
num_features = len(feature_names)
num_classes = len(class_idx)

# Create train and test features as a numpy array.
x_train = train_data[feature_names].to_numpy()
x_test = test_data[feature_names].to_numpy()
# Create train and test targets as a numpy array.
y_train = train_data["subject"]
y_test = test_data["subject"]
```

Реализуем референсную модель и распечатаем спраку по её слоям.

```
def create_baseline_model(hidden_units, num_classes, dropout_rate=0.2):
    inputs = layers.Input(shape=(num_features,), name="input_features")
    x = create_ffn(hidden_units, dropout_rate, name=f"ffn_block1")(inputs)
    for block_idx in range(4):
        # Create an FFN block.
        x1 = create_ffn(hidden_units, dropout_rate,
                name=f"ffn_block{block_idx + 2}")(x)
        # Add skip connection.
        x = layers.Add(name=f"skip_connection{block_idx + 2}")([x, x1])
    # Compute logits.
    logits = layers.Dense(num_classes, name="logits")(x)
    # Create the model.
    return keras.Model(inputs=inputs, outputs=logits, name="baseline")

baseline_model = create_baseline_model(hidden_units,
                                       num_classes, dropout_rate)
baseline_model.summary()
```

Проведем обучение референсной модели на наших данных.

```
history = run_experiment(baseline_model, x_train, y_train)
```

Визуализируем результаты обучения модели.

```
display_learning_curves(history)
```

Проверим качество предсказаний референсной модели на тестовом датасете.

```
_, test_accuracy = baseline_model.evaluate(x=x_test, y=y_test, verbose=0)
print(f"Test accuracy: {round(test_accuracy * 100, 2)}%")
```

1.4. Построение графовой нейронной сети. Подготовка данных для обучения графовой нейронной сети в общем случае является сложной задачей. Мы задействуем одну из самых простых схем: (node_features, edges, edge_weights).

```
# Список ребер для цитирований (размер [2, num_edges]).
edges = citations[["source", "target"]].to_numpy().T
# Веса в нашей модели отсутсвуют, поэтому зададим их единицами.
edge_weights = tf.ones(shape=edges.shape[1])
# Создадим список для вершин и их признаков [num_nodes, num_features].
node_features = tf.cast(
    papers.sort_values("paper_id")[feature_names].to_numpy(),
    dtype=tf.dtypes.float32
)
# Итоговая структура для описания графа/подграфа в нашей модели
graph_info = (node_features, edges, edge_weights)

print("Edges shape:", edges.shape)
print("Nodes shape:", node_features.shape)
```

Предусмотрим вспомогательную функцию для подключения простого рекуррентного слоя.

```
def create_gru(hidden_units, dropout_rate):
    inputs = keras.layers.Input(shape=(2, hidden_units[0]))
    x = inputs
    for units in hidden_units:
      x = layers.GRU(
          units=units,
          activation="tanh",
          recurrent_activation="sigmoid",
          return_sequences=True,
          dropout=dropout_rate,
          return_state=False,
          recurrent_dropout=dropout_rate,
      )(x)
    return keras.Model(inputs=inputs, outputs=x)
```

Базовый сверточный графовый нейронный слой в формате Keras.

```
class GraphConvLayer(layers.Layer):
    def __init__(
        self,
        hidden_units,
        dropout_rate=0.2,
```

```
        aggregation_type="mean",
        combination_type="concat",
        normalize=False,
        *args,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)

        self.aggregation_type = aggregation_type
        self.combination_type = combination_type
        self.normalize = normalize
        self.ffn_prepare = create_ffn(hidden_units, dropout_rate)
        if self.combination_type == "gru":
            self.update_fn = create_gru(hidden_units, dropout_rate)
        else:
            self.update_fn = create_ffn(hidden_units, dropout_rate)

    def prepare(self, node_repesentations, weights=None):
        # node_repesentations имеет размер [num_edges, embedding_dim].
        messages = self.ffn_prepare(node_repesentations)
        if weights is not None:
            messages = messages * tf.expand_dims(weights, -1)
        return messages

    def aggregate(self, node_indices,
                        neighbour_messages,
                        node_repesentations):
        # node_indices имеет размер [num_edges].
        # neighbour_messages имеет размер: [num_edges, representation_dim].
        # node_repesentations имеет размер [num_nodes, representation_dim].
        num_nodes = node_repesentations.shape[0]
        if self.aggregation_type == "sum":
            aggregated_message = tf.math.unsorted_segment_sum(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        elif self.aggregation_type == "mean":
            aggregated_message = tf.math.unsorted_segment_mean(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        elif self.aggregation_type == "max":
            aggregated_message = tf.math.unsorted_segment_max(
                neighbour_messages, node_indices, num_segments=num_nodes
            )
        else:
            raise ValueError(f"Invalid aggregation type:
                                {self.aggregation_type}.")
```

```
        return aggregated_message

    def update(self, node_repesentations, aggregated_messages):
        # node_repesentations имеет размер [num_nodes, representation_dim].
        # aggregated_messages имеет размер [num_nodes, representation_dim].
        if self.combination_type == "gru":
            # Создаем последовательность для слоя GRU
            h = tf.stack([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "concat":
            h = tf.concat([node_repesentations, aggregated_messages], axis=1)
        elif self.combination_type == "add":
            h = node_repesentations + aggregated_messages
        else:
            raise ValueError(f"Invalid combination type:
                                {self.combination_type}.")

        # Применяем на выбор GRU или FFN
        node_embeddings = self.update_fn(h)
        if self.combination_type == "gru":
            node_embeddings = tf.unstack(node_embeddings, axis=1)[-1]
        if self.normalize:
            node_embeddings = tf.nn.l2_normalize(node_embeddings, axis=-1)
        return node_embeddings

    def call(self, inputs):
        """
        Обработка входных данных для получения вложения node_embeddings.
        inputs: тройки вида: node_repesentations, edges, edge_weights.
        Returns: node_embeddings размера [num_nodes, representation_dim].
        """
        node_repesentations, edges, edge_weights = inputs
        # node_indices (source) и neighbour_indices (target) из ребер.
        node_indices, neighbour_indices = edges[0], edges[1]
        # neighbour_repesentations размера [num_edges, representation_dim].
        neighbour_repesentations = tf.gather(node_repesentations,
                                             neighbour_indices)
        # Подготовим данные для аггрегации на основе информации об окружении
        neighbour_messages = self.prepare(  neighbour_repesentations,
                                            edge_weights)
        # Проведем аггрегацию в соответсвии с выбранным алгоритмом.
        aggregated_messages = self.aggregate(
            node_indices, neighbour_messages, node_repesentations
        )
        # Обновим информацию о вложении вершин
        return self.update(node_repesentations, aggregated_messages)
```

Подготовим итоговый класс для графовой нейронной сети, задействовав:

- Предобработка с помощью FFN для получения начального представления вершин.
- Подключение нескольких сверточных графовых слоев с опциональными сквозными соединениями.
- Постобработка с помощью FFN и пребразование к формату logit.

```
class GNNNodeClassifier(tf.keras.Model):
    def __init__(
        self,
        graph_info,
        num_classes,
        hidden_units,
        aggregation_type="sum",
        combination_type="concat",
        dropout_rate=0.2,
        normalize=True,
        *args,
        **kwargs,
    ):
        super().__init__(*args, **kwargs)
        node_features, edges, edge_weights = graph_info
        self.node_features = node_features
        self.edges = edges
        self.edge_weights = edge_weights
        # Set edge_weights to ones if not provided.
        if self.edge_weights is None:
            self.edge_weights = tf.ones(shape=edges.shape[1])
        # Scale edge_weights to sum to 1.
        self.edge_weights = self.edge_weights /
            tf.math.reduce_sum(self.edge_weights)
        # Слой для предобработки
        self.preprocess = create_ffn(hidden_units,
                            dropout_rate, name="preprocess")
        # Первый сверточный слой
        self.conv1 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
            combination_type,
            normalize,
            name="graph_conv1",
        )
        # Второй сверточный слой
        self.conv2 = GraphConvLayer(
            hidden_units,
            dropout_rate,
            aggregation_type,
```

```
            combination_type,
            normalize,
            name="graph_conv2",
        )
        # Слой для постобработки
        self.postprocess = create_ffn(hidden_units,
                            dropout_rate, name="postprocess")
        # Create a compute logits layer.
        self.compute_logits = layers.Dense(units=num_classes,
                            name="logits")

    def call(self, input_node_indices):
        x = self.preprocess(self.node_features)
        x1 = self.conv1((x, self.edges, self.edge_weights))
        # Простейшая сквозная связь
        x = x1 + x
        x2 = self.conv2((x, self.edges, self.edge_weights))
        # Простейшая сквозная связь
        x = x2 + x
        x = self.postprocess(x)
        node_embeddings = tf.gather(x, input_node_indices)
        # Compute logits
        return self.compute_logits(node_embeddings)
```

Создадим тестовую реализацию графовой сверточной сети и выведем по ней краткую справку.

```
gnn_model = GNNNodeClassifier(
    graph_info=graph_info,
    num_classes=num_classes,
    hidden_units=hidden_units,
    dropout_rate=0.2,
    name="gnn_model",
)
print("GNN output shape:", gnn_model(input_node_indices=[1,10,100]))
gnn_model.summary()
```

1.5. Обучение и оценка качества модели. Проведем обучение графовой нейронной модели на тех же настройках, что мы применяли для референсной модели.

```
x_train = train_data.paper_id.to_numpy()
history = run_experiment(gnn_model, x_train, y_train)
```

Оценим качество её работы и сравним с графиками для референсной модели.

```
display_learning_curves(history)
```

С высокой вероятностью наша модель выдаст худщие результаты, чем референсная.

§ 2. Предсказание свойств молекулярных графов

Предустановим все необходимые пакеты для работы с молекулярными графами, опционально обновив использующие их библиотеки.

```
!pip -q install rdkit-pypi
!pip -q install numpy
!pip -q install pandas
!pip -q install Pillow
!pip -q install matplotlib
!pip -q install pydot
!sudo apt-get -qq install graphviz
!pip -q install tensorflow
```

Подключим все библиотеки и проверим, что код исполняется без ошибок.

```
import os

# Temporary suppress tf logs
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import warnings
from rdkit import Chem
from rdkit import RDLogger
from rdkit.Chem.Draw import IPythonConsole
from rdkit.Chem.Draw import MolsToGridImage

# Temporary suppress warnings and RDKit logs
warnings.filterwarnings("ignore")
RDLogger.DisableLog("rdApp.*")

np.random.seed(42)
tf.random.set_seed(42)
```

Загрузим базу данных: названия молекул, молекулярные формулы и параметр проницаемости гемоэнцефалического барьера для этих молекул. Наша задача будет состоять в бинарной классификации молекулярных графов — на основе их формулы сделать вывод способны они или нет пройти через гемоэнцефалический барьер (проникать в головной мозг).

```
csv_path = keras.utils.get_file(
    "BBBP.csv", "https://deepchemdata.s3-us-west-
    1.amazonaws.com/datasets/BBBP.csv"
)
df = pd.read_csv(csv_path, usecols=[1, 2, 3])
```

```
df.iloc[96:104]
```

Для кодирования свойств атомов и связей между ними мы предусмотрим два класса: AtomFeaturizer и BondFeaturizer.

```
class Featurizer:
    def __init__(self, allowable_sets):
        self.dim = 0
        self.features_mapping = {}
        for k, s in allowable_sets.items():
            s = sorted(list(s))
            self.features_mapping[k] = dict(zip(s,range(self.dim,
                                                    len(s) + self.dim)))
            self.dim += len(s)

    def encode(self, inputs):
        output = np.zeros((self.dim,))
        for name_feature, feature_mapping in self.features_mapping.items():
            feature = getattr(self, name_feature)(inputs)
            if feature not in feature_mapping:
                continue
            output[feature_mapping[feature]] = 1.0
        return output

class AtomFeaturizer(Featurizer):
    def __init__(self, allowable_sets):
        super().__init__(allowable_sets)

    def symbol(self, atom):
        return atom.GetSymbol()

    def n_valence(self, atom):
        return atom.GetTotalValence()

    def n_hydrogens(self, atom):
        return atom.GetTotalNumHs()

    def hybridization(self, atom):
        return atom.GetHybridization().name.lower()

class BondFeaturizer(Featurizer):
    def __init__(self, allowable_sets):
        super().__init__(allowable_sets)
        self.dim += 1

    def encode(self, bond):
        output = np.zeros((self.dim,))
```

```
        if bond is None:
            output[-1] = 1.0
            return output
        output = super().encode(bond)
        return output

    def bond_type(self, bond):
        return bond.GetBondType().name.lower()

    def conjugated(self, bond):
        return bond.GetIsConjugated()

atom_featurizer = AtomFeaturizer(
    allowable_sets={
        "symbol": {"B", "Br", "C", "Ca", "Cl", "F", "H", "I", "N",
        "Na", "O", "P", "S"},
        "n_valence": {0, 1, 2, 3, 4, 5, 6},
        "n_hydrogens": {0, 1, 2, 3, 4},
        "hybridization": {"s", "sp", "sp2", "sp3"},
    }
)
bond_featurizer = BondFeaturizer(
    allowable_sets={
        "bond_type": {"single", "double", "triple", "aromatic"},
        "conjugated": {True, False},
    }
)
```

Подготовим три функции для работы с молекулярными графами:

- molecule_from_smiles — на основе молекулярной нотации SMILES выдавать молекулярный объект пакета rdkit.
- graph_from_molecule — на основе молекулярного объекта пакета rdkit выдавать граф для данной молекулы.
- graph_from_smiles —на основе молекулярной нотации SMILES выдавать граф для данной молекулы.

```
def molecule_from_smiles(smiles):

    molecule = Chem.MolFromSmiles(smiles, sanitize=False)

    # If sanitization is unsuccessful, catch the error, and try again
    # without the sanitization step that caused the error
    flag = Chem.SanitizeMol(molecule, catchErrors=True)
    if flag != Chem.SanitizeFlags.SANITIZE_NONE:
        Chem.SanitizeMol(molecule,
                         sanitizeOps=Chem.SanitizeFlags.SANITIZE_ALL ^ flag)
    Chem.AssignStereochemistry(molecule, cleanIt=True, force=True)
    return molecule
```

```
def graph_from_molecule(molecule):
    # Initialize graph
    atom_features = []
    bond_features = []
    pair_indices = []

    for atom in molecule.GetAtoms():
        atom_features.append(atom_featurizer.encode(atom))

        # Add self-loops
        pair_indices.append([atom.GetIdx(), atom.GetIdx()])
        bond_features.append(bond_featurizer.encode(None))

        for neighbor in atom.GetNeighbors():
            bond = molecule.GetBondBetweenAtoms(atom.GetIdx(),
                                                neighbor.GetIdx())
            pair_indices.append([atom.GetIdx(), neighbor.GetIdx()])
            bond_features.append(bond_featurizer.encode(bond))

    return np.array(atom_features),
            np.array(bond_features),
            np.array(pair_indices)

def graphs_from_smiles(smiles_list):
    # Initialize graphs
    atom_features_list = []
    bond_features_list = []
    pair_indices_list = []

    for smiles in smiles_list:
        molecule = molecule_from_smiles(smiles)
        atom_features, bond_features, pair_indices =
                                    graph_from_molecule(molecule)

        atom_features_list.append(atom_features)
        bond_features_list.append(bond_features)
        pair_indices_list.append(pair_indices)

    return (
        tf.ragged.constant(atom_features_list, dtype=tf.float32),
        tf.ragged.constant(bond_features_list, dtype=tf.float32),
        tf.ragged.constant(pair_indices_list, dtype=tf.int64),
    )
```

Разобьем базу данных на обучающую, валидатационную и тестовую выборки и преобразуем входные данные к формату молекулярных графов.

```
# Shuffle array of indices ranging from 0 to 2049
```

```
permuted_indices = np.random.permutation(np.arange(df.shape[0]))

# Train set: 80 % of data
train_index = permuted_indices[: int(df.shape[0] * 0.8)]
x_train = graphs_from_smiles(df.iloc[train_index].smiles)
y_train = df.iloc[train_index].p_np

# Valid set: 19 % of data
valid_index = permuted_indices[int(df.shape[0] * 0.8) : int(df.shape[0] * 0.99
x_valid = graphs_from_smiles(df.iloc[valid_index].smiles)
y_valid = df.iloc[valid_index].p_np

# Test set: 1 % of data
test_index = permuted_indices[int(df.shape[0] * 0.99) :]
x_test = graphs_from_smiles(df.iloc[test_index].smiles)
y_test = df.iloc[test_index].p_np
```

Проверим работу наших функций, визуализировав молекулярный граф.

```
print(f"Name:\t{df.name[100]}\nSMILES:\t{df.smiles[100]}
            \nBBBP:\t{df.p_np[100]}")
molecule = molecule_from_smiles(df.iloc[100].smiles)
print("Molecule:")
molecule
```

Поскольку у молекулярных графов в нашей нотации будут присутсвовать петли, то для рассмотренного примера число ребер должно быть равно сумме числа реальных ребер и числа вершин.

```
graph = graph_from_molecule(molecule)
print("Graph (including self-loops):")
print("\tatom features\t", graph[0].shape)
print("\tbond features\t", graph[1].shape)
print("\tpair indices\t", graph[2].shape)
```

Для подготовки пакета (батча) из молекулярных графов нам потребуется объединить их в большой глобальный граф. Отдельная компонента связности такого глобального графа — это отдельный молекулярный граф из батча.

```
def prepare_batch(x_batch, y_batch):
    """Merges (sub)graphs of batch into a single global graph
    """
    atom_features, bond_features, pair_indices = x_batch

    # Obtain number of atoms and bonds for each graph (molecule)
    num_atoms = atom_features.row_lengths()
    num_bonds = bond_features.row_lengths()

    # Obtain partition indices (molecule_indicator), which will be used to
    # gather (sub)graphs from global graph in model later on
    molecule_indices = tf.range(len(num_atoms))
    molecule_indicator = tf.repeat(molecule_indices, num_atoms)
```

```
    # Merge (sub)graphs into a global (disconnected) graph.
    # Adding 'increment' to 'pair_indices' (and merging ragged tensors)
    # actualizes the global graph
    gather_indices = tf.repeat(molecule_indices[:-1], num_bonds[1:])
    increment = tf.cumsum(num_atoms[:-1])
    increment = tf.pad(tf.gather(increment, gather_indices),
                       [(num_bonds[0], 0)])
    pair_indices = pair_indices.merge_dims(outer_axis=0,
                                           inner_axis=1).to_tensor()
    pair_indices = pair_indices + increment[:, tf.newaxis]
    atom_features = atom_features.merge_dims(outer_axis=0,
                                           inner_axis=1).to_tensor()
    bond_features = bond_features.merge_dims(outer_axis=0,
                                           inner_axis=1).to_tensor()

    return (atom_features,\
            bond_features,\
            pair_indices,\
            molecule_indicator),\
            y_batch
```

На основе нашей функции для подготовки батчей реализуем итоговый метод для генерации датасетов.

```
def MPNNDataset(X, y, batch_size=32, shuffle=False):
    dataset = tf.data.Dataset.from_tensor_slices((X, (y)))
    if shuffle:
        dataset = dataset.shuffle(1024)
    return dataset.batch(batch_size).map(prepare_batch, -1).prefetch(-1)
```

2.1. Реализация модели графовой нейронной сети. В качестве основы для нашей модели выберем модель из работы Neural Message Passing for Quantum Chemistry. Собственно, передача сообщений в такой графовой нейронной сети осуществляется в два этапа:

- Реберная сеть передает 'сообщения' каждой вершине v от её ближайшего окружения. Эти сообщения зависят от свойств (feutures) ребер, которые соединяют вершины и приводят к пересчету текущего состояния v.
- Простейшая рекуррентная сеть (GRU) получает на входе информацию о предыдущем состоянии вершины v, что в свою очередь позволяет запоминать эти состояния и динамически обновлять их в памяти сети.

```
class EdgeNetwork(layers.Layer):
    def build(self, input_shape):
        self.atom_dim = input_shape[0][-1]
        self.bond_dim = input_shape[1][-1]
        self.kernel = self.add_weight(
            shape=(self.bond_dim, self.atom_dim * self.atom_dim),
            initializer="glorot_uniform",
```

```
            name="kernel",
        )
        self.bias = self.add_weight(
            shape=(self.atom_dim * self.atom_dim,),
            initializer="zeros",
            name="bias",
        )
        self.built = True

    def call(self, inputs):
        atom_features, bond_features, pair_indices = inputs

        # Apply linear transformation to bond features
        bond_features = tf.matmul(bond_features, self.kernel) + self.bias

        # Reshape for neighborhood aggregation later
        bond_features = tf.reshape(bond_features, (-1, self.atom_dim,
                                                   self.atom_dim))

        # Obtain atom features of neighbors
        atom_features_neighbors = tf.gather(atom_features,
                                            pair_indices[:, 1])
        atom_features_neighbors = tf.expand_dims(atom_features_neighbors,
                                                 axis=-1)

        # Apply neighborhood aggregation
        transformed_features = tf.matmul(bond_features,
                                         atom_features_neighbors)
        transformed_features = tf.squeeze(transformed_features, axis=-1)
        aggregated_features = tf.math.unsorted_segment_sum(
            transformed_features,
            pair_indices[:, 0],
            num_segments=tf.shape(atom_features)[0],
        )
        return aggregated_features

class MessagePassing(layers.Layer):
    def __init__(self, units, steps=4, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.steps = steps

    def build(self, input_shape):
        self.atom_dim = input_shape[0][-1]
        self.message_step = EdgeNetwork()
        self.pad_length = max(0, self.units - self.atom_dim)
        self.update_step = layers.GRUCell(self.atom_dim + self.pad_length)
```

```
        self.built = True

    def call(self, inputs):
        atom_features, bond_features, pair_indices = inputs

        # Pad atom features if number of desired units
        #  exceeds atom_features dim.
        atom_features_updated = tf.pad(atom_features, [(0, 0),
                                       (0,self.pad_length)])

        # Perform a number of steps of message passing
        for i in range(self.steps):
            # Aggregate information from neighbors
            atom_features_aggregated = self.message_step(
                [atom_features_updated, bond_features, pair_indices]
            )
            # Update node state via a step of GRU
            atom_features_updated, _ = self.update_step(
                atom_features_aggregated, atom_features_updated
            )
        return atom_features_updated
```

После окончания процедуры обмена сообщениями мы будем:

- Инфомацию о состоянии вершин мы разобъем на компоненты связности, соответсвующие отдельным молекулярным графам из батча.
- Каждый из этих графов мы дополним нулями до максимального в рамках батча.
- Предусмотрим маскирование для добавленных нулевых вершин.
- Итоговый тензор с инфомацией о вершинах передается трансфомеру.

```
class PartitionPadding(layers.Layer):
    def __init__(self, batch_size, **kwargs):
        super().__init__(**kwargs)
        self.batch_size = batch_size

    def call(self, inputs):

        atom_features, molecule_indicator = inputs

        # Obtain subgraphs
        atom_features_partitioned = tf.dynamic_partition(
            atom_features, molecule_indicator, self.batch_size
        )

        # Pad and stack subgraphs
        num_atoms = [tf.shape(f)[0] for f in atom_features_partitioned]
        max_num_atoms = tf.reduce_max(num_atoms)
        atom_features_stacked = tf.stack(
```

```
            [
                tf.pad(f, [(0, max_num_atoms - n), (0, 0)])
                for f, n in zip(atom_features_partitioned, num_atoms)
            ],
            axis=0,
        )

        # Remove empty subgraphs (usually for last batch in dataset)
        gather_indices =
            tf.where(tf.reduce_sum(atom_features_stacked,(1, 2)) != 0)
        gather_indices = tf.squeeze(gather_indices, axis=-1)
        return tf.gather(atom_features_stacked, gather_indices, axis=0)
class TransformerEncoderReadout(layers.Layer):
    def __init__(
        self, num_heads=8, embed_dim=64,
        dense_dim=512, batch_size=32, **kwargs
    ):
        super().__init__(**kwargs)

        self.partition_padding = PartitionPadding(batch_size)
        self.attention = layers.MultiHeadAttention(num_heads, embed_dim)
        self.dense_proj = keras.Sequential(
            [layers.Dense(dense_dim, activation="relu"),
            layers.Dense(embed_dim),]
        )
        self.layernorm_1 = layers.LayerNormalization()
        self.layernorm_2 = layers.LayerNormalization()
        self.average_pooling = layers.GlobalAveragePooling1D()

    def call(self, inputs):
        x = self.partition_padding(inputs)
        padding_mask = tf.reduce_any(tf.not_equal(x, 0.0), axis=-1)
        padding_mask = padding_mask[:, tf.newaxis, tf.newaxis, :]
        attention_output = self.attention(x, x, attention_mask=padding_mask)
        proj_input = self.layernorm_1(x + attention_output)
        proj_output = self.layernorm_2(proj_input +
                                       self.dense_proj(proj_input))
        return self.average_pooling(proj_output)
```

Итоговая модель будет собираться из уже определеных нами блоков, а также двух дополнительных слоев для решения итоговой задачи бинарной классификации.

```
def MPNNModel(
    atom_dim,
    bond_dim,
    batch_size=32,
    message_units=29,
```

```
    message_steps=4,
    num_attention_heads=8,
    dense_units=512,
):

    atom_features = layers.Input((atom_dim,), dtype="float32",
                                 name="atom_features")
    bond_features = layers.Input((bond_dim,), dtype="float32",
                                 name="bond_features")
    pair_indices = layers.Input((2,), dtype="int32", name="pair_indices")
    molecule_indicator = layers.Input((), dtype="int32",
                                 name="molecule_indicator")

    x = MessagePassing(message_units, message_steps)(
        [atom_features, bond_features, pair_indices]
    )

    x = TransformerEncoderReadout(
        num_attention_heads, message_units, dense_units, batch_size
    )([x, molecule_indicator])

    x = layers.Dense(dense_units, activation="relu")(x)
    x = layers.Dense(1, activation="sigmoid")(x)

    model = keras.Model(
        inputs=[atom_features,
            bond_features,
            pair_indices,
            molecule_indicator],
        outputs=[x],
    )
    return model
```

Создадим тестовую версию для нашей модели и визуализируем её.

```
mpnn = MPNNModel(
    atom_dim=x_train[0][0][0].shape[0], bond_dim=x_train[1][0][0].shape[0],
    dense_units = 128
)
mpnn.compile(
    loss=keras.losses.BinaryCrossentropy(),
    optimizer=keras.optimizers.Adam(learning_rate=5e-4),
    metrics=[keras.metrics.AUC(name="AUC")],
)
keras.utils.plot_model(mpnn, show_dtype=True, show_shapes=True)
```

Сформируем датасеты и проведем тестовое обучение итоговой модели.

```
train_dataset = MPNNDataset(x_train, y_train)
valid_dataset = MPNNDataset(x_valid, y_valid)
```

```
test_dataset = MPNNDataset(x_test, y_test)

history = mpnn.fit(
    train_dataset,
    validation_data=valid_dataset,
    epochs=40,
    verbose=2,
    class_weight={0: 2.0, 1: 0.5},
)

plt.figure(figsize=(10, 6))
plt.plot(history.history["AUC"], label="train AUC")
plt.plot(history.history["val_AUC"], label="valid AUC")
plt.xlabel("Epochs", fontsize=16)
plt.ylabel("AUC", fontsize=16)
plt.legend(fontsize=16)
```

§ 3. Практические задания

Задание № 1: Доработка сверточной графовой нейронной модели.

1. Доработать нашу модель: меняя параметры, типы аггрегации и комбинации, количество слоев свертки, значения для dropout.
2. Добиться, чтобы графовая модель стабильно выдавала лучший результат, чем референсная с такими же параметрами (dropout, batch_size).

Задание № 2: Доработка графовой модели для молекулярных графов.

1. Попробовать обучать модель на графических ускорителях. Сравнить время обучения графовой модели на ускорителях и на процессорах.
2. Доработать нашу модель: меняя параметры, добавляя слои dropout и меняя их вероятности.
3. Добиться, чтобы графовая модель стабильно выдавала результат на тестовых и валидатационных данных выше 95 процентов.