神经信息传递法—预测分子和材料特性

前言

> 基于图神经网络（**GNN**）的方法在模拟复杂相互作用方面具有巨大的潜力，被广泛应用于分子量子力学性质以及材料预测的预测。目前为止，传统的机器学习模型普遍应用于预测分子性质，与**GNN**不同，**ML**模型需要先对分子特征进行操作，尽管这些分子特征是各种分子性质的良好预测指标，不过基于网络训练的模型，效果可能会更好。

1.消息传递神经网络（MPNN）

近年来，随着量子化学计算和分子动力学模拟等实验展开了巨大的数据量，图神经网络能够应用于分子模型（原子系统的对称性），**MPNN**正是可以应用于图上的监督学习框架。在图中，所有数据节点相互连接，意味着数据不再独立，这使得大多数标准机器学习模型无法使用。图神经网络可以从图形中提取数值数据或使用直接对此类数据进行操作的模型。这种架构可以从图形中提取数值数据或使用直接对此类数据进行操作的模型

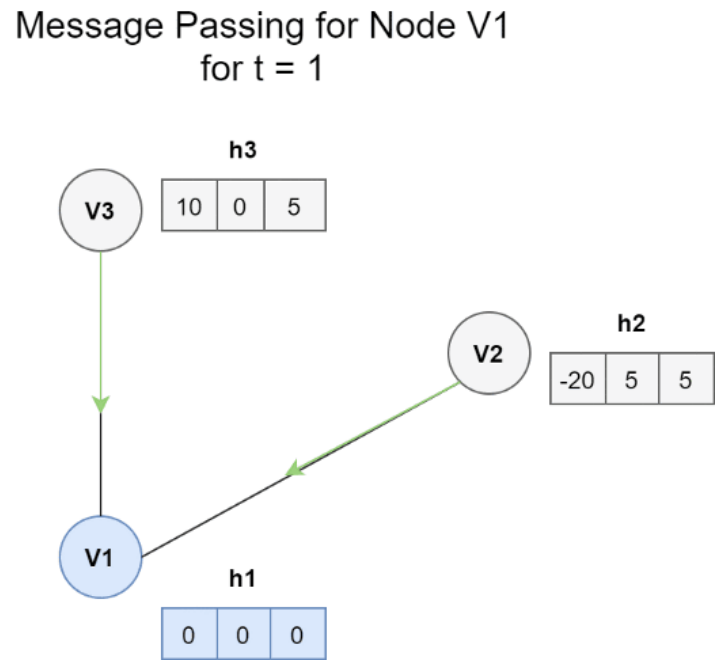
1.1 MPNN概念

MPNN概念很简单，图中的每个节点都有一个隐藏状态（即特征向量）。对于每个节点**V**t，我们将所有相邻节点的隐藏状态和可能的边与节点**V**t本身聚合在一起。然后，我们使用获得的消息和该节点先前的隐藏状态来更新节点**V**t的隐藏状态

$$m_v^{t+1} = \sum_{w \in N(v)} h_w^t$$

$$h_v^{t+1} = average(h_v, m_v^{t+1})$$

ht - hidden state for each node



注：**节点V1**的消息传递架构的一个非常简单的示例。在这种情况下，消息是邻居隐藏状态的总和。更新函数是消息**m**和**h1**之间的平均值（图参考该网站）

主要有三个方程定义了**MPNN**框架。从相邻节点获得的消息由以下等式给出：

$$m_v^{t+1} = \sum_{w \in N(v)} M_t \left(h_v^t, h_w^t, e_{vw}\right)$$

它是从邻居节点获得所有消息**Mt**的总和。**Mt**是一个任意函数，它取决于相邻节点的隐藏状态和边缘。我们可以通过保留一些输入参数来简化这个函数。使用一个方程来更新节点**Vt**的隐藏状态：

$$h_v^{t+1} = U_t \left(h_v^t, m_v^{t+1}\right)$$

节点**Vt**的隐藏状态是通过用新获得的消息**Mv**更新旧的隐藏状态获得的。从上图可以了解，更新函数**Ut**是之前隐藏状态与消息的平均值。我们重复这个消息传递算法指定的次数。最后建立读出阶段公式：

$$\hat{y} = R\left(\left\{h_v^T \mid v \in G\right\}\right)$$

在该式，我们可以提取所有更新的隐藏状态并创建一个描述整个图的最终特征向量。然后将该特征向量作为标准机器学习模型的输入。

2. MPNN用于分子特性预测

参考**keras**官方案例将通过**MPNN**来预测血脑屏障通透性（**blood-brain barrier permeability**）的分子特性。数据集将从[MoleculeNet.org](https://moleculenet.org/datasets-1)下载。

2.1 数据集

该数据包含**2050**个分子。每个分子都有对应的名称，标签和**SMILES**字符串。**BBB**是将血液与脑细胞外液隔开的膜，因此阻止了大多数药物（分子）到达大脑。正因为如此，**BBBP**对于研究针对中枢神经系统的新药的开发非常重要。该数据集的标签是二进制（**0或1**），表示分子的渗透性。

首先导入相关库：

```
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)
```

下载数据集，并展示部分；

```
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]
```

	name	p_np	smiles
96	cefoxitin	1	CO[C@]1(NC(=O)Cc2sccc2)[C@H]3SCC(=C(N3C1=O)C(O...
97	Org34167	1	NC(CC=C)c1ccccc1c2noc3c2cccc3
98	9-OH Risperidone	1	OC1C(N2CCC1)=NC(C)=C(CCN3CCC(CC3)c4c5ccc(F)cc5...
99	acetaminophen	1	CC(=O)Nc1ccc(O)cc1
100	acetylsalicylate	0	CC(=O)Oc1ccccc1C(O)=O
101	allopurinol	0	O=C1N=CN=C2NNC=C12
102	Alprostadil	0	CCCCC[C@H](O)/C=C/[C@H]1[C@H](O)CC(=O)[C@@H]1C...
103	aminophylline	0	CN1C(=O)N(C)c2nc[nH]c2C1=O.CN3C(=O)N(C)c4nc[nH...

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2050 entries, 0 to 2049
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   name    2050 non-null   object
 1   p_np    2050 non-null   int64 
 2   smiles  2050 non-null   object
dtypes: int64(1), object(2)
memory usage: 48.2+ KB
```

2.2 定义特征

为了编码原子和键的原子特征。我们将定义两个类：**AtomFeaturizer**和**BondFeaturizer**。仅考虑少数（原子和键）特征：[原子特征]符号（元素），价电子，氢键数，轨道杂交，[键特征]（共价）键类型和共轭。

```
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},
    }
)
```

2.3 生成图表

在外面从**SMILES**生成完整的图之前，我们需要实现以下功能：

1. `molecule_from_smiles`，它将一个 SMILES 作为输入并返回一个分子对象。这一切都由 RDKit 处理。
2. `graph_from_molecule`，它将分子对象作为输入并返回一个图，表示为一个三元组（atom_features、bond_features、pair_indices）。为此，我们将使用之前定义的类。

最后，我们可以实现**graphs_from_smiles**，它将函数（1）（2）应用于训练，验证，和测试数据集的所有**SMILES**。

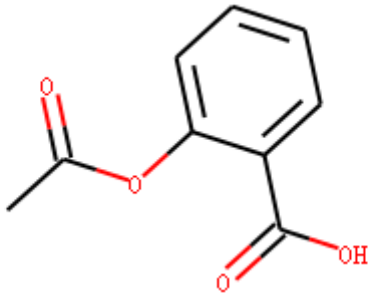
```
def molecule_from_smiles(smiles):
    # MolFromSmiles(m, sanitize=True) 应该等同于
    # MolFromSmiles(m, sanitize=False) -> SanitizeMol(m) ->
AssignStereochemistry(m, ...)
    molecule = Chem.MolFromSmiles(smiles, sanitize=False)

    # If sanitization 不成功，捕捉错误，在尝试一次
    # 捕捉错误的步骤
    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):
    # 初始化图
    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)

    # 将列表转换为张量，用于tf.data.Dataset
    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),
    )


# 清洗数组的指数范围从0-2049
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
```

3.4 测试函数

```
#测试功能
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
```

Name: acetylsalicylate
SMILES: CC(=O)Oc1ccccc1C(O)=O
BBBP: 0
Molecule:



查看生成图的结构：

```
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)
```

```
Graph (including self-loops):
	atom features	 (13, 29)
	bond features	 (39, 7)
	pair indices	 (39, 2)
```

3.5 创建tf.data.Dataset

在该教程中**MPNN**实现将作为单个图作为输入（每次迭代）。因此，给定一批子图分子，我们需要将它们合并为一个图（称之为全局图）。全局图是一个断开的图，其中每个子图都与其他子图完全分离。

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

    atom_features, bond_features, pair_indices = x_batch

    # 获得每个图形（分子）的原子和键的数量
    num_atoms = atom_features.row_lengths()
    num_bonds = bond_features.row_lengths()

    #获得分区指数（molecule_indicator），这些指数将被用于以后从模型中的全局图中收集（子）图
    molecule_indices = tf.range(len(num_atoms))
    molecule_indicator = tf.repeat(molecule_indices, num_atoms)

    # 将（子）图合并成一个全局的（不相连的）图。增加'增量'topair_indices'（并合并锯齿状的张量），使全局图实际化
    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


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)
```

3.6 MPNN模型构建

MPNN模型设计基于Neural Message Passing for Quantum Chemistry和 DeepChem 的 MPNNModel实现一个 MPNN。模型内容包括上述三个公式（阶段）：消息传递，读出和分类。

消息传递的步骤本身由两部分组成：

1. 边缘网络，它根据它们之间的边缘特征从**w_{i}**的临节点传递信息**v**，从而更新节点。表示的邻居**v'**。
2. **GRU**，使用门作为最近节点的输入，并根据之前的节点状态对其进行更新。通俗来讲，最近的节点状态作为**GRU**的输入，而先前的节点状态被合并到**GRU**的内存状态中。这允许信息从一个节点状态传播到另一个节点状态。

重复，1，2步骤，并且每个步骤中的**1 ... k**,聚合信息的半径从**v**增加到**1**

```
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

        # bond features线性转换
        bond_features = tf.matmul(bond_features, self.kernel) + self.bias

        # 重构临节点聚合
        bond_features = tf.reshape(bond_features, (-1, self.atom_dim,
self.atom_dim))

        # 获得临界点原子特征
        atom_features_neighbors = tf.gather(atom_features, pair_indices[:, 1])
        atom_features_neighbors = tf.expand_dims(atom_features_neighbors,
axis=-1)

        # 应用临节点聚合
        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

        # 如果所需单位的数量超过atom_features dim，则对原子特征进行填充
        # 这里可以使用一个密集层。
        atom_features_updated = tf.pad(atom_features, [(0, 0), (0,
self.pad_length)])

        # 消息传递步骤迭代
        for i in range(self.steps):
            # 汇集临节点信息
            atom_features_aggregated = self.message_step(
                [atom_features_updated, bond_features, pair_indices]
            )

            # 通过GRU更新节点状态
            atom_features_updated, _ = self.update_step(
                atom_features_aggregated, atom_features_updated
            )
        return atom_features_updated
```

当消息传递过程结束时，**k-step-aggregated**节点状态被划分为子图（对应于批次中的每个分子），然后减少到图级嵌入。在论文中，为此目的使用了一个**set-to-set**层。在**keras**教程中，使用**transformer encoder**+平均池化的方法。具体来说；

- **k**步聚合节点状态被划分为子图
- 填充每个子图以匹配具有最大节点数的子图，其次是**tf.stack(...)**
- 堆叠填充张量，屏蔽编码子图确保填充不会干扰训练
- 将张量传递给**transformer encoder**，最后平均池化

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

    def call(self, inputs):
```

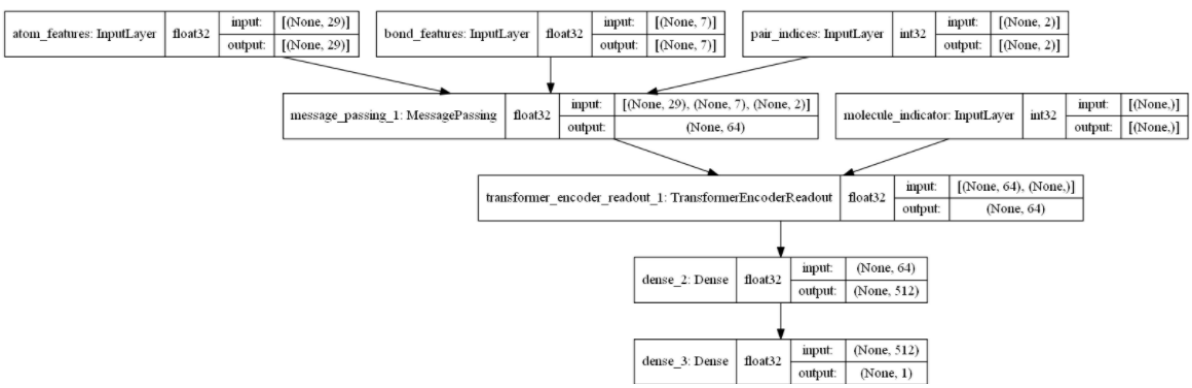
```
        atom_features, molecule_indicator = inputs

        # 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)
```

除了消息传递和读出外，还将实施一个两层分类网络来对**BBBP**进行预测

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

```
    message_units=64,
    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


mpnn = MPNNModel(
    atom_dim=x_train[0][0][0].shape[0], bond_dim=x_train[1][0][0].shape[0],
)

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)
```

我们将整个**MPNN**模型的个节点以及对应节点的特征类型绘制出来，如下图所示；



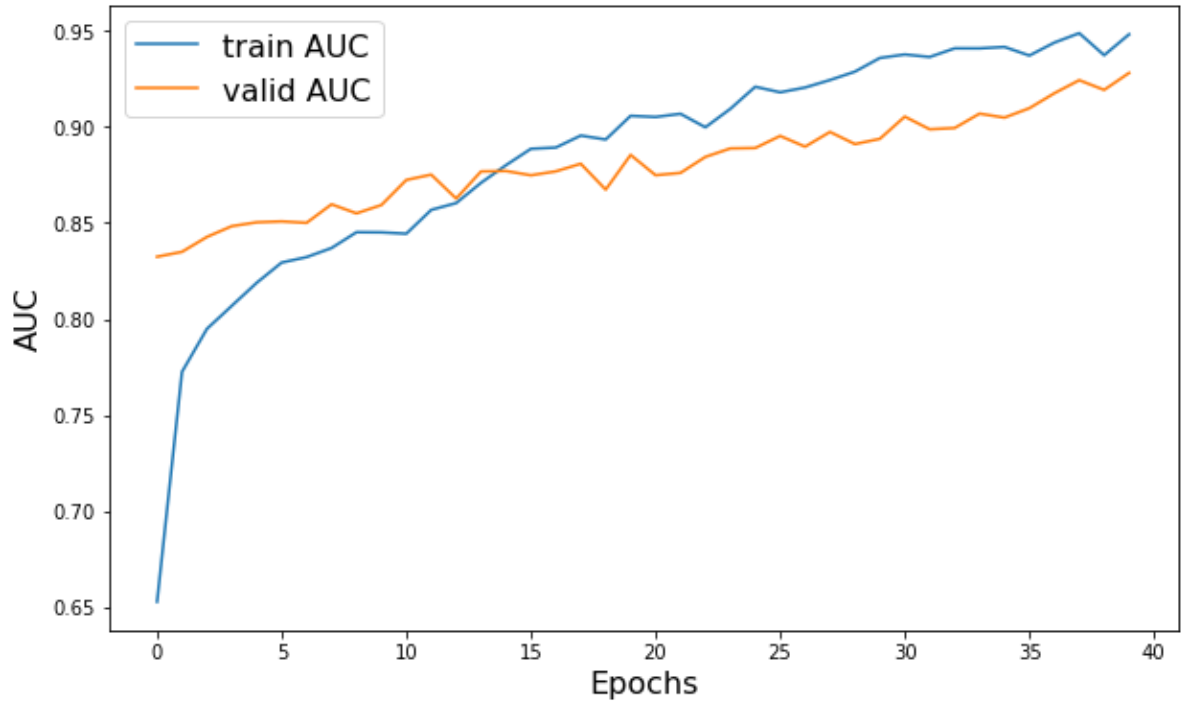
开始训练模型，在验证模型我们选择输出每次**epoch**的详细状态，选择{**0: 2.0, 1: 0.5**}权重值用于加权损失函数（因为样本数不够有可能会出现代表性不足的情况）

```
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)p
```

训练结果如下所示；

```
52/52 - 16s - loss: 0.3117 - AUC: 0.9180 - val_loss: 0.4697 - val_AUC: 0.8955
Epoch 27/40
52/52 - 15s - loss: 0.3067 - AUC: 0.9205 - val_loss: 0.5844 - val_AUC: 0.8897
Epoch 28/40
52/52 - 14s - loss: 0.2992 - AUC: 0.9244 - val_loss: 0.4637 - val_AUC: 0.8974
Epoch 29/40
52/52 - 14s - loss: 0.2891 - AUC: 0.9288 - val_loss: 0.5016 - val_AUC: 0.8911
Epoch 30/40
52/52 - 14s - loss: 0.2751 - AUC: 0.9359 - val_loss: 0.5672 - val_AUC: 0.8938
Epoch 31/40
52/52 - 14s - loss: 0.2705 - AUC: 0.9377 - val_loss: 0.4293 - val_AUC: 0.9054
Epoch 32/40
52/52 - 14s - loss: 0.2736 - AUC: 0.9364 - val_loss: 0.4051 - val_AUC: 0.8988
Epoch 33/40
52/52 - 14s - loss: 0.2647 - AUC: 0.9409 - val_loss: 0.4293 - val_AUC: 0.8994
Epoch 34/40
52/52 - 14s - loss: 0.2634 - AUC: 0.9409 - val_loss: 0.4626 - val_AUC: 0.9069
Epoch 35/40
52/52 - 14s - loss: 0.2605 - AUC: 0.9416 - val_loss: 0.3689 - val_AUC: 0.9048
Epoch 36/40
52/52 - 14s - loss: 0.2716 - AUC: 0.9371 - val_loss: 0.3521 - val_AUC: 0.9097
Epoch 37/40
52/52 - 14s - loss: 0.2571 - AUC: 0.9439 - val_loss: 0.3233 - val_AUC: 0.9175
Epoch 38/40
52/52 - 14s - loss: 0.2449 - AUC: 0.9488 - val_loss: 0.3118 - val_AUC: 0.9243
Epoch 39/40
52/52 - 14s - loss: 0.2706 - AUC: 0.9373 - val_loss: 0.3977 - val_AUC: 0.9193
Epoch 40/40
52/52 - 14s - loss: 0.2445 - AUC: 0.9482 - val_loss: 0.3341 - val_AUC: 0.9281
```

将每次**epoch**的**AUC**曲线绘制出来如下图所示；



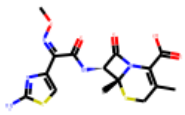
我们可以观察到，一共40次**epoch**曲线尽管小范围有波动，总整体观察看呈上升趋势，如果增加**epoch**次数很有可能增加测试和预测的正确率（有兴趣的可以增加**epoch**次数，本文不再增加）。

3.7 测试模型效果

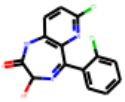
rdkit库中提供了一些方法可以返回模型的图，绘制预测结果如下；

```
#预测
molecules = [molecule_from_smiles(df.smiles.values[index]) for  index in
test_index]
y_true = [df.p_np.values[index] for index in test_index]
y_pred = tf.squeeze(mpnn.predict(test_dataset), axis=1)

legends = [f"y_true/y_pred = {y_true[i]}/{y_pred[i]:.2f}" for   i in
range(len(y_true))]
MolsToGridImage(molecules, molsPerRow=4, legends=legends)
```

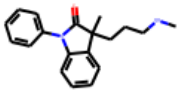
结果如下图所示；



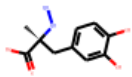
y_true/y_pred = 0/0.00



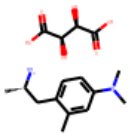
y_true/y_pred = 1/0.83



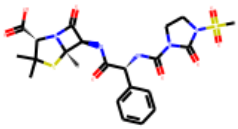
y_true/y_pred = 1/0.99



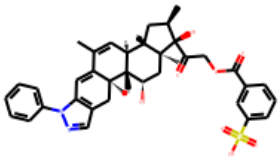
y_true/y_pred = 0/0.01



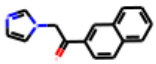
y_true/y_pred = 1/0.01



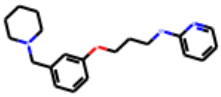
y_true/y_pred = 0/0.00



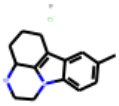
y_true/y_pred = 1/0.90



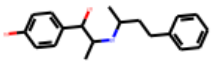
y_true/y_pred = 1/0.83



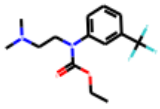
y_true/y_pred = 1/0.97



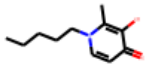
y_true/y_pred = 1/0.97



y_true/y_pred = 0/0.08

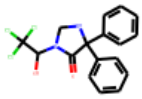


y_true/y_pred = 1/0.99

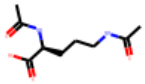


y_true/y_pred = 1/0.88

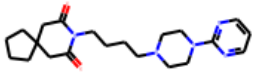
y_true/y_pred = 1/0.94



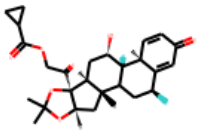
y_true/y_pred = 1/0.97



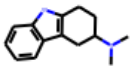
y_true/y_pred = 1/0.60



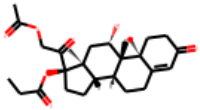
y_true/y_pred = 1/0.98



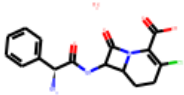
y_true/y_pred = 1/0.97



y_true/y_pred = 1/0.99



y_true/y_pred = 1/0.97



y_true/y_pred = 0/0.00

4 结论

本章节的核心内容是信息传递模型的基本理论和实例演练，我们可以看到没有经过修改的模型在预测不同分子的血脑屏障通透性方面效果显著。从**SMILES**构建图，然后图上运行模型，最后训练预测，为我们下一步学习基于**MPNN**模型延申的前沿模型提供了理论基础和参考。有兴趣的朋友可以参考更多实例在[keras网站]

参考文献

1. Message-passing neural network (MPNN) for molecular property prediction
2. Message Passing Neural Network
3. brain-research
4. tf.keras.layers.MultiHeadAttention方法

推荐阅读

- 微分算子法
- 使用PyTorch构建神经网络模型进行手写识别
- 使用PyTorch构建神经网络模型以及反向传播计算
- 如何优化模型参数，集成模型
- TORCHVISION目标检测微调教程
- 神经网络开发食谱
- 主成分分析（PCA）方法步骤以及代码详解
- 神经网络编码分类变量—categorical_embdedder
- 多层感知机预测算法—以二手车预测为例（MLP)