深度学习解决方案

前面特征构造可以直接加入不用管，加入自己的特征就行

导入模块

> 建议在云服务器上尝试运行。
> 免费算力资源可以在Kaggle Notebook (30h/week)、Google Colab、Azure上找到
> 付费的推荐恒源云，一小时三毛钱可以玩玩，崩溃了切换别的服务器轻松无负担
> 最后做多个模型的融合就行，前提是模型性能别差太多，深度学习和机器学习可以直接融合

```
import numpy as np
import pandas as pd
import copy
import gc
import time
from pandas.tseries.holiday import USFederalHolidayCalendar as calendar
from sklearn.decomposition import PCA

import tqdm
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.cluster import DBSCAN,KMeans
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error as MSE
from sklearn.preprocessing import LabelEncoder
encoder = LabelEncoder()

from sklearn.ensemble import RandomForestRegressor
import xgboost as xgb
from xgboost import plot_importance, plot_tree
import lightgbm as lgb

import torch
import torch.nn as nn
from torch.utils.data import DataLoader,Dataset
from torch.nn import functional as F

from joblib import dump, load
%matplotlib inline
```

读取数据

> Addition 和 Weather 是后续加的，不需要或者要引入别的直接导入就行

```
train=pd.read_csv("path/train.zip")
test=pd.read_csv("path/test.zip")

B=pd.read_csv("path/Addition.csv")
train=pd.concat([train,B],axis=1)
B=pd.read_csv("path/Addition_test.csv")
test=pd.concat([test,B],axis=1)
```

聚类特征但是没有用到

```
# coord=train[['pickup_latitude','pickup_longitude']].values
# coord1=test[['pickup_latitude','pickup_longitude']].values
# c=np.vstack((coord,coord1))
# # 下车点聚类
# # 不做DBSCAN了，有机会再试
# coord=train[['dropoff_latitude','dropoff_longitude']].values
# coord1=test[['dropoff_latitude','dropoff_longitude']].values
# d=np.vstack((coord,coord1))
# f=np.vstack((c,d))
# Cluster=KMeans(n_clusters=18, random_state=0, n_init=15).fit(f)
# dump(Cluster,"cluster.pkl")
```

保存后直接读取结果就是了

```
test_C=load("cluster.pkl")
train['in_Cluster']=test_C.predict(train[['pickup_latitude','pickup_longitude']].values)
train['out_Cluster']=test_C.predict(train[['dropoff_latitude','dropoff_longitude']].values)
test['in_Cluster']=test_C.predict(test[['pickup_latitude','pickup_longitude']].values)
test['out_Cluster']=test_C.predict(test[['dropoff_latitude','dropoff_longitude']].values)
```

天气数据预处理

```
weather = pd.read_csv("wdn2016.csv")
weather['precipitation'] = pd.to_numeric(weather['precipitation'], errors='coerce')
weather['snow fall'] = pd.to_numeric(weather['snow fall'], errors='coerce')
```

```
weather['snow depth'] = pd.to_numeric(weather['snow depth'], errors='coerce')
weather = weather.fillna(0)
```

⚠ **特征工程**

建议统一放在一个函数里面，方便调用接口

```
def process(train,is_Train=True):


##############################################################################
############################
    ############                          时间特征
################################

##############################################################################
############################

    def rush_hour_f(row):
        rhour = row['real_hour']
        if (6 <= rhour) & (rhour <= 10):
            return 1
        if (10 < rhour) & (rhour < 16):
            return 2
        if (16 <= rhour) & (rhour <= 20):
            return 3
        return 0

    encoder.fit(train['store_and_fwd_flag'])
    train['store_and_fwd_flag'] =
encoder.transform(train['store_and_fwd_flag'])

    train['pickup_datetime'] = pd.to_datetime(train['pickup_datetime'])
    train['month'] = train['pickup_datetime'].dt.month
    train['weekday'] = train['pickup_datetime'].dt.weekday
    train['hour'] = train['pickup_datetime'].dt.hour
    train['minute'] = train['pickup_datetime'].dt.minute
    train['minute_of_day'] = train['hour'] * 60 + train['minute']
    train['real_hour'] = train['minute_of_day'] / 60
        #for weather
    train['year'] = train['pickup_datetime'].dt.year
    train['day'] = train['pickup_datetime'].dt.day
    train['rush_hour'] = train.apply(rush_hour_f, axis=1)

    cal = calendar()
    holidays = cal.holidays(start=pd.Timestamp(2015,12,31),
end=pd.Timestamp(2017,1,1))
    H=[[i.month,i.day] for i in holidays]
    train['is_weekend']=train['weekday'].apply(lambda x:1 if x>4 else 0)
    for i,j in H:
```

```
        # 这里特别值得注意，查询返回的是一个视图
        # 对视图的修改并不会返回到原始上
        # 因此，我们需要做的是，采用iloc获得条件查询的index并进行修改
        train.iloc[train[(train['month']==i)&(train['day']==j)].index,-1]=1
    encoder.fit(train['is_weekend'])
    train['is_weekend'] = encoder.transform(train['is_weekend'])


################################################################################
############################
    ############                        空间特征
################################

################################################################################
############################
    AVG_EARTH_RADIUS =6369
    def ft_haversine_distance(lat1, lng1, lat2, lng2):
    # Haversine距离，用于测量大地距离
        lat1, lng1, lat2, lng2 = map(np.radians, (lat1, lng1, lat2, lng2))
        lat = lat2 - lat1
        lng = lng2 - lng1
        d = np.sin(lat * 0.5) ** 2 + np.cos(lat1) * np.cos(lat2) * np.sin(lng
* 0.5) ** 2
        h = 2 * AVG_EARTH_RADIUS * np.arcsin(np.sqrt(d))
        return h

    def ft_degree(lat1,log1,lat2,log2):
        # 用于计算坐标方位
        lng_delta_rad = np.radians(log2 - log1)
        lat1, lng1, lat2, lng2 = map(np.radians, (lat1,log1,lat2,log2))
        y = np.sin(lng_delta_rad) * np.cos(lat2)
        x = np.cos(lat1) * np.sin(lat2) - np.sin(lat1) * np.cos(lat2) *
np.cos(lng_delta_rad)
        return np.degrees(np.arctan2(y, x))
    train['distance']=ft_haversine_distance(train['pickup_latitude'].values,

train['pickup_longitude'].values,

train['dropoff_latitude'].values,

train['dropoff_longitude'].values)
    train['direction'] = ft_degree(train['pickup_latitude'].values,
                                   train['pickup_longitude'].values,
                                   train['dropoff_latitude'].values,
                                   train['dropoff_longitude'].values)

    if is_Train:
        train = train[(train.trip_duration < 1000000)]
        train = train[train['pickup_longitude'].between(-75, -73)]
        train = train[train['pickup_latitude'].between(40, 42)]
        train = train[train['dropoff_longitude'].between(-75, -73)]
        train = train[train['dropoff_latitude'].between(40, 42)]
        duration = train['trip_duration']
```

```
        train['speed'] = train.distance / duration * 2236.936292
        train['trip_duration'] = np.log(train['trip_duration'].values)


################################################################################
############################
    ############                          附带信息特征
################################

################################################################################
############################
        #NYS max speed limit 55mph

    weather["date"] = pd.to_datetime(weather["date"],format='%d-%m-%Y')
    weather['year'] = weather['date'].dt.year
    weather_2016 = weather[weather["year"] == 2016]
    weather_2016.drop(["year"], axis=1, inplace=True)


train['date']=pd.to_datetime(train[['year','month','day']],errors='coerce')
    left_merge = pd.merge(left=train, right=weather_2016, on="date",
how="left")
    train = left_merge.loc[:, left_merge.columns != 'date']

    train['in_Cluster']=train['in_Cluster'].astype(str)
    train['out_Cluster']=train['out_Cluster'].astype(str)
    train['node']=train['in_Cluster']+"/"+train['out_Cluster']

    encoder.fit(train['node'])
    train['node']=encoder.transform(train['node'])
    encoder.fit(train['in_Cluster'])
    train['in_Cluster']=encoder.transform(train['in_Cluster'])
    encoder.fit(train['out_Cluster'])
    train['out_Cluster']=encoder.transform(train['out_Cluster'])

    return train
```

```
train,test=process(train),process(test,False) # 这样就好了
```

数据清洗

```
x_var=train[train.speed<37]
y_var=x_var.trip_duration
```

按理说就是这样，你想要什么数据就直接这样处理，注意获取的结果要么是深拷贝形式，要么换一个变量记录，这样甚至就可以控制变量来探究数据清洗过程的好坏，找到一个比较合理的数据清洗力度。

深度学习模型

比较复杂的模型：UNet-Transformer 1D

```
class ResBlock(nn.Module):
    def __init__(self, in_channels, filters, output_filters=None):
        super(ResBlock, self).__init__()

        if output_filters is None:
            output_filters = filters
        self.conv1 = nn.Conv1d(in_channels, filters, kernel_size=3,
padding='same')

        self.group_norm1 = nn.BatchNorm1d(filters)
        self.silu1 = nn.SiLU()

        self.group_norm2 = nn.BatchNorm1d(output_filters)
        self.silu2 = nn.SiLU()
        self.conv2 = nn.Conv1d(filters, output_filters, kernel_size=3,
padding='same')

        # 如果输入和卷积输出的通道数不同，调整输入的通道数以匹配
        if in_channels != output_filters:
            self.shortcut = nn.Conv1d(in_channels, output_filters,
kernel_size=1, padding='same')
        else:
            self.shortcut = nn.Identity()

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.group_norm1(out)
        out = self.silu1(out)

        out = self.conv2(out)
        out = self.group_norm2(out)
        out = self.silu2(out)

        # Apply the shortcut if necessary
        residual = self.shortcut(residual)

        out += residual
        return out

class TransformerBlock(nn.Module):
    def __init__(self, input_dim, num_heads, ff_dim, dropout=0.2):
        super(TransformerBlock, self).__init__()
        self.multi_head_attention =
nn.MultiheadAttention(embed_dim=input_dim, num_heads=num_heads,
dropout=dropout)
```

```
        self.layer_norm1 = nn.LayerNorm(input_dim)
        self.dropout1 = nn.Dropout(dropout)

        self.ffn = nn.Sequential(
            nn.Linear(input_dim, ff_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(ff_dim, input_dim)
        )
        self.layer_norm2 = nn.LayerNorm(input_dim)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x):
        # Multi-head attention
        attn_output, _ = self.multi_head_attention(x, x, x)
        attn_output = self.dropout1(attn_output)
        out1 = self.layer_norm1(x + attn_output)

        # Feedforward network
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output)
        out2 = self.layer_norm2(out1 + ffn_output)
        return out2



class UNet1D_res(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()

        # 下采样编码器
        self.enc1 = nn.ModuleList(
            [ResBlock(in_ch, 128),
             ResBlock(128, 128)]
        )
        self.enc2 = nn.ModuleList(
            [ResBlock(128, 256),
             ResBlock(256, 256)]
        )
        self.enc3 = nn.ModuleList(
            [ResBlock(256, 256),
             ResBlock(256, 256)]
        )
        self.enc4=nn.ModuleList(
            [ResBlock(256,256) for i in range(2)]
        )

        self.bottleneck=TransformerBlock(256,64,512)
        # 池化
        self.pool = nn.MaxPool1d(2)

        # 上采样解码器
        self.up1 = nn.ConvTranspose1d(256, 256, kernel_size=2, stride=2)
```

```
        self.dec1 = nn.ModuleList(
            [ResBlock(512, 256),
             ResBlock(256, 256),
             ResBlock(256, 256),
             ]
        )


        self.dec2 = nn.ModuleList(
            [ResBlock(512, 256),
             ResBlock(256, 256),
             ResBlock(256, 256)
             ]
        )
        self.dec3 = nn.ModuleList(
            [ResBlock(512, 256),
             ResBlock(256, 256),
             ResBlock(256, 256)
             ]
        )
        self.dec4 = nn.ModuleList(
            [ResBlock(384, 256),
             ResBlock(256, 256),
             ResBlock(256, 256)
             ]
        )

        self.head = nn.Sequential(
            ResBlock(256, 128),
             ResBlock(128, 64),
             ResBlock(64, 32),
             ResBlock(32, 1),

        )

        # 输出层
        self.linear = nn.Linear(16, 1)

    def forward(self, x):
        enc1 = x
        for layer in self.enc1:
            enc1 = layer(enc1)

        enc2 = self.pool(enc1)
        for layer in self.enc2:
            enc2 = layer(enc2)

        enc3 = self.pool(enc2)
        for layer in self.enc3:
            enc3 = layer(enc3)

        enc4=self.pool(enc3)
        for layr in self.enc4:
```

```
            enc4=layr(enc4)

        enc4=enc4.transpose(2,1)

        bottleneck=self.bottleneck(enc4)

        dec1=torch.concat([bottleneck,enc4],dim=2)
        dec1=dec1.transpose(2,1)


        for ly in self.dec1:
            dec1=ly(dec1)

        dec1=self.up1(dec1)
        dec2=torch.concat([dec1,enc3],dim=1)


        for ly in self.dec2:
            dec2 = ly(dec2)
        dec2=self.up1(dec2)

        dec3=torch.concat([dec2,enc2],dim=1)

        for ly in self.dec3:
            dec3 = ly(dec3)
        dec3=self.up1(dec3)

        dec4=torch.concat([dec3,enc1],dim=1)

        for ly in self.dec4:
            dec4 = ly(dec4)

        head=self.head(dec4)

        out=self.linear(head.squeeze())

        return out
```

这种模型需要将数据转换为序列：

```
def x_to_seq(x):
    s=x.shape[1]
    x_seq0=x[:,s-16:].reshape(-1,1,16)
    # 添加其他序列的属性
    x_seq1=x[:,:s-16].reshape(-1,s-16,1).repeat(1,1,16)

    # 获得第一个序列(-1,4,1)
    # 将其沿axis=1轴重复n次
    # (-1,4,40)
```

```
    # 按维度拼接
    return torch.cat((x_seq0,x_seq1),axis=1)
```

整个流程走下来就是：

```
def train_(model,train_Data,path="1.pth",epoch=200):

    criterion=nn.MSELoss()
    # optimizer=torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9,
weight_decay=1e-5, nesterov=True)
    optimizer =torch.optim.AdamW(model.parameters(), lr=0.01)

    device="cuda" if torch.cuda.is_available() else "cpu"
    # device="cpu"
    if torch.cuda.device_count() > 1:
        print(f"Let's use {torch.cuda.device_count()} GPUs!")
        model = nn.DataParallel(model)

    model.to(device)
    criterion.to(device)
    Best_=np.inf
    for e in tqdm.tqdm(range(epoch)):
        model.train()
        total_loss=0
        for idx,(x,y) in enumerate(train_Data):

            x,y=x.to(device),y.to(device)
            optimizer.zero_grad()
            y_pre=model(x)
            loss=torch.sqrt(criterion(y_pre,y))
            loss.backward()
            optimizer.step()
            total_loss+=loss.item()


        if (total_loss)< Best_:
            Best_=total_loss
            state={
                "model_state_dict":model.state_dict()
                }
            torch.save(state,path)

        print(f"epoch: {e+1}/{epoch}")
        print(f"Score: {total_loss/len(train_Data)}")

class LoadData(Dataset):
    def __init__(self,x,y):
        self.x=x
        self.y=y

    def __getitem__(self, idx):
```

```
        return self.x[idx],self.y[idx]

    def __len__(self):
        return  len(self.x)

std=StandardScaler()

feature_forTrain=[ 'pickup_longitude', 'pickup_latitude',
       'dropoff_longitude', 'dropoff_latitude', 'vendor_id',
       'pc', 'store_and_fwd_flag',
       'month', 'weekday',
        'rush_hour', 'is_weekend',
          'distance','humidity', 'pressure', 'temperature',
       'wind_direction', 'wind_speed',
       'direction', 'precipitation', 'snow fall', 'snow depth'
         ]
 x_to_train=copy.deepcopy(x_var[feature_forTrain])

def get_dummy(data,col):
    val=pd.get_dummies(data[col],dtype=np.uint8,prefix=col)
    data=pd.concat([data,val],axis=1)
    data.drop(col,axis=1,inplace=True)
    return data

oneHot=[
    'vendor_id',
    'store_and_fwd_flag',
    'rush_hour',
    'is_weekend',
    'pc',
    'month',
    'weekday'
]

for i in oneHot:
    x_to_train=get_dummy(x_to_train,i)

x_train=std.fit_transform(x_to_train)
x_train_tensor=torch.from_numpy(x_train).float()

mean_y,std_y=y_var.mean(),y_var.std()
y_=(y_var-mean_y)/std_y
y_train_tensor=torch.from_numpy(y_.values).float()

y_train_tensor=y_train_tensor.reshape(-1,1)

def x_to_seq(x):
    s=x.shape[1]
    x_seq0=x[:,s-16:].reshape(-1,1,16)
    # 添加其他序列的属性
    x_seq1=x[:,:s-16].reshape(-1,s-16,1).repeat(1,1,16)

    # 获得第一个序列(-1,4,1)
```

```
    # 将其沿axis=1轴重复n次
    # (-1,4,40)
    # 按维度拼接
    return torch.cat((x_seq0,x_seq1),axis=1)

x_seq=x_to_seq(x_train_tensor)

train_Data_Unet=DataLoader(
    dataset=LoadData(x_seq,y_train_tensor),
    batch_size=1500,
    shuffle=True,
    drop_last=True
)

m1=UNet1D_res(x_seq.shape[1],1)

train_(m1,train_Data_Unet,path="model/dl1.pth",epoch=150)
m1.load_state_dict(torch.load("model/dl1.pth")['model_state_dict'])
m1.to("cpu")


test_for_dl=test[feature_forTrain]
for i in oneHot:
    test_for_dl=get_dummy(test_for_dl,i)
test_for_dl=std.fit_transform(test_for_dl)
test_for_dl=torch.from_numpy(test_for_dl).float()
y_pred=m1(test_for_dl)
def create_submission(y_pred,name='sub'):
    y_pred=y_pred*std_y+mean_y
    submission = pd.DataFrame({'id': test.id, 'trip_duration':
np.exp(y_pred.detach().numpy().reshape(-1))})
    submission.fillna( submission['trip_duration'].mean(),inplace=True)
    submission.to_csv("%s.csv"%name,index=False)
    return submission

create_submission(y_pred).describe()
```

结果好不好另说，起码这个很酷。

改成其他小模型训练速度会快很多，甚至几个数量级。贴几个可以用的，直接更改模型和参数就行，比如：

```
m=CNN(x_seq.shape[1])
```

非卷积模型就不需要转为序列了，MLP直接全参数输入即可。

Model 1 MLP1

```
class MLP1(nn.Module):
    def __init__(self, in_dim):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(in_dim, 256),
            nn.LayerNorm(256),
            nn.GELU(),
            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.GELU(),
            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.GELU(),
            nn.Linear(64, 32),
            nn.GELU(),
            nn.Linear(32, 20),
            nn.GELU(),
            nn.Linear(20, 18),
            nn.GELU(),
            nn.Linear(18, 16),
            nn.GELU(),
            nn.Linear(16, 14),
            nn.GELU(),
            nn.Linear(14, 12),
            nn.GELU(),
            nn.Linear(12, 10),
            nn.GELU(),
            nn.Linear(10, 8),
            nn.GELU(),
            nn.Linear(8, 6),
            nn.GELU(),
            nn.Linear(6, 4),
            nn.GELU(),
            nn.Linear(4, 2),
            nn.GELU(),
            nn.Linear(2, 1),

        )

    def forward(self, x):
        return self.layers(x)
```

Model 2 LSTM1

```
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim=64, output_dim=1, num_layers=2):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, num_layers,
batch_first=True)
        self.fc = nn.Sequential(
```

```
            nn.Linear(hidden_dim, 128),
            nn.BatchNorm1d(128),
            nn.GELU(),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.GELU(),
            nn.Linear(256, 16),
            nn.GELU(),
            nn.Linear(16, 1),

        )


    def forward(self, x):
        # LSTM需要的输入是 (batch_size, seq_length, num_features)
        lstm_out, _ = self.lstm(x)
        # 仅使用最后一个时间步的输出
        out = self.fc(lstm_out[:, -1, :])
        return out
```

Model 3 MLP2

```
class NetR(nn.Module):
    def
__init__(self,in_features=48,n_hidden1=48,n_hidden2=64,n_hidden3=32,out_featu
res=1):
        super(NetR, self).__init__()
        self.flatten=nn.Flatten()
        self.hidden1=nn.Sequential(
            nn.Linear(in_features,n_hidden1,False),
            nn.BatchNorm1d(n_hidden1),
            nn.GELU()
        )
        self.hidden2=nn.Sequential(
            nn.Linear(in_features,n_hidden1),
            nn.Dropout(0.5),
            nn.BatchNorm1d(n_hidden1),
           nn.GELU()
        )
        self.hidden3=nn.Sequential(
            nn.Linear(n_hidden1,n_hidden2),
            nn.BatchNorm1d(n_hidden2),
            nn.GELU()
        )
        self.hidden5=nn.Sequential(
            nn.Linear(n_hidden2,n_hidden3),
            nn.BatchNorm1d(n_hidden3),
            nn.GELU()

        )
        self.out=nn.Sequential(nn.Linear(n_hidden3,out_features))
```

```
    def forward(self,x):
        x1=self.hidden1(x)
        x2=self.hidden2(x)
        x3=self.hidden3(x2+x1)

        o=self.hidden5(x3)

        return self.out(o)
```

Model 4 UNet1

```
class Conv1D_Block(nn.Module):
    def __init__(self,in_ch,out_ch,kernel_size=3,padding=1):
        super().__init__()

self.conv=nn.Conv1d(in_ch,out_ch,kernel_size=kernel_size,padding=padding)
        self.bn=nn.BatchNorm1d(out_ch)
        self.relu=nn.GELU()
    def forward(self,x):
        return self.relu(self.bn(self.conv(x)))

class UNet1D(nn.Module):
    def __init__(self,in_ch,out_ch):
        super().__init__()

        # 下采样编码器
        self.enc1=Conv1D_Block(in_ch,64)
        self.enc2=Conv1D_Block(64,128)
        self.enc3=Conv1D_Block(128,256)

        # 池化
        self.pool=nn.MaxPool1d(2)

        # 上采样解码器
        self.up1=nn.ConvTranspose1d(256,128,kernel_size=2,stride=2)
        self.dec1=Conv1D_Block(256,128)
        self.up2=nn.ConvTranspose1d(128,64,kernel_size=2,stride=2)
        self.dec2=Conv1D_Block(128,64)

        # 全连接输出
        self.out=nn.Linear(64,out_ch)

    def forward(self,x):
        enc1=self.enc1(x)
        enc2=self.enc2(self.pool(enc1))
        enc3=self.enc3(self.pool(enc2))

        dec1=self.up1(enc3)
        dec1=torch.cat((enc2,dec1),dim=1)
```

```
        dec1=self.dec1(dec1)

        dec2 = self.up1(dec1)
        dec2 = torch.cat((enc1, dec2), dim=1)
        dec2 = self.dec2(dec2)
        return self.out(dec2)
```

Model 5 ResNet1

```
class ResBlock(nn.Module):
    def __init__(self, in_channels, filters, output_filters=None, groups=8):
        super(ResBlock, self).__init__()

        if output_filters is None:
            output_filters = filters

        self.group_norm1 = nn.GroupNorm(num_groups=groups,
num_channels=in_channels)
        self.silu1 = nn.SiLU()
        self.conv1 = nn.Conv1d(in_channels, filters, kernel_size=3,
padding='same')

        self.group_norm2 = nn.GroupNorm(num_groups=groups,
num_channels=filters)
        self.silu2 = nn.SiLU()
        self.conv2 = nn.Conv1d(filters, output_filters, kernel_size=3,
padding='same')

        # 如果输入和卷积输出的通道数不同，调整输入的通道数以匹配
        if in_channels != output_filters:
            self.shortcut = nn.Conv1d(in_channels, output_filters,
kernel_size=1, padding='same')
        else:
            self.shortcut = nn.Identity()

    def forward(self, x):
        residual = x

        out = self.group_norm1(x)
        out = self.silu1(out)
        out = self.conv1(out)

        out = self.group_norm2(out)
        out = self.silu2(out)
        out = self.conv2(out)

        # Apply the shortcut if necessary
        residual = self.shortcut(residual)

        out += residual
```

```
        return out

def repeat_block(x,filters,repeat):
    for _ in range(repeat):
        x=ResBlock(x,filters)

# Example usage
x = torch.randn(10, 64, 100)  # (batch_size, channels, length)
model = ResBlock(in_channels=64, filters=128)
output = model(x)
print(output.shape)
```

Model 6 UNet2

```
class ResBlock(nn.Module):
    def __init__(self, in_channels, filters, output_filters=None, groups=8):
        super(ResBlock, self).__init__()

        if output_filters is None:
            output_filters = filters
        self.conv1 = nn.Conv1d(in_channels, filters, kernel_size=3,
padding='same')

        self.group_norm1 = nn.BatchNorm1d(filters)
        self.silu1 = nn.SiLU()

        self.group_norm2 = nn.BatchNorm1d(output_filters)
        self.silu2 = nn.SiLU()
        self.conv2 = nn.Conv1d(filters, output_filters, kernel_size=3,
padding='same')

        # 如果输入和卷积输出的通道数不同，调整输入的通道数以匹配
        if in_channels != output_filters:
            self.shortcut = nn.Conv1d(in_channels, output_filters,
kernel_size=1, padding='same')
        else:
            self.shortcut = nn.Identity()

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.group_norm1(out)
        out = self.silu1(out)

        out = self.conv2(out)
        out = self.group_norm2(out)
        out = self.silu2(out)

        # Apply the shortcut if necessary
```

```
        residual = self.shortcut(residual)

        out += residual
        return out

class UNet1D_res(nn.Module):
    def __init__(self,in_ch,out_ch):
        super().__init__()

        # 下采样编码器
        self.enc1=ResBlock(in_ch,64)
        self.enc2=ResBlock(64,128)
        self.enc3=ResBlock(128,256)

        # 池化
        self.pool=nn.MaxPool1d(2)

        # 上采样解码器
        self.up1=nn.ConvTranspose1d(256,128,kernel_size=2,stride=2)
        self.dec1=ResBlock(256,128)
        self.up2=nn.ConvTranspose1d(128,64,kernel_size=2,stride=2)
        self.dec2=ResBlock(128,64)

        # 输出层
        self.final_conv = nn.Conv1d(64, out_ch, kernel_size=1)
        self.linear=nn.Linear(40,1)

    def forward(self,x):
        enc1=self.enc1(x)
        enc2=self.enc2(self.pool(enc1))
        enc3=self.enc3(self.pool(enc2))


        dec1=self.up1(enc3)

        dec1=torch.cat((enc2,dec1),dim=1)
        dec1=self.dec1(dec1)

        dec2 = self.up2(dec1)
        dec2 = torch.cat((enc1, dec2), dim=1)
        dec2 = self.dec2(dec2)
        # final_conv: [b,1,40]
        out=self.final_conv(dec2).squeeze(1)

        return self.linear(out)
```