流量预测

0. Github

> https://github.com/Liyanhao1209/BDL

branch:lab1

1. 实验要求

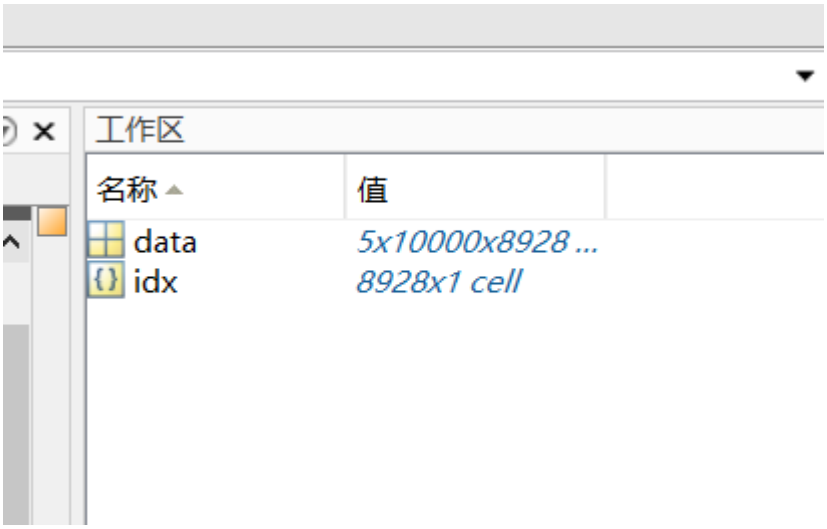
1. 将数据根据时间索引划分成训练集（70%）和测试集（30%），比如有 10天的数据，那么就选择前7天的数据作为训练集，后3天的数据作为测试集；
2. 对5045小区的测试集进行预测，可以先不运行在Hadoop上，而是以scikit learn 工具包进行机器学习；

这里我没有选择scikitlearn工具包，而是选用了**tensorflow+keras构建CNN卷积神经网络**进行流量预测。

2. 数据集预处理：重塑、划分、再重塑

2.1. 怎么解决频繁加载数据集到内存的问题？

这个问题之前还没怎么困扰到我，因为之前的数据分析大部分是用matlab做的，这款软件的优势就在于执行脚本读数据集后，数据集一直就会驻留在内存里。



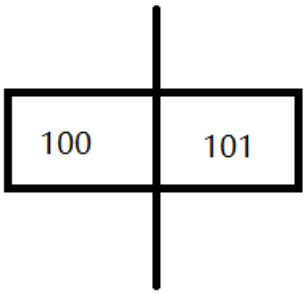
但是python不行，python脚本执行结束会自动GC，下一次还得重新装载数据集到内存。

所以我选择了**Jupyter Notebook**，他相当于在计算机的一个端口上启动一个服务，服务只要在运行，就相当于加载过的数据集一直在内存中

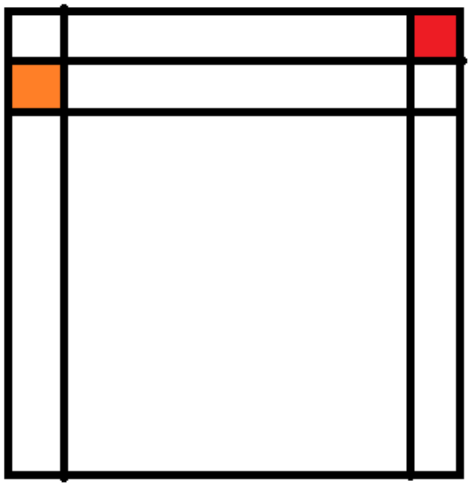
2.2. 重塑原始数据集

我们知道，原始数据集是8928*10000*5的形状。其中第二个维度是小区编号（一维）。

不过，一维的编号反应不出物理位置信息，比如一维编号100和101，是相邻的。但是映射到二维，他俩分别是(1,100)和(2,1)，隔着十万八千里。



一维相邻



二维不相邻

为了保留下来这个位置信息，我们需要重塑数据集，从(8928,10000,5) → (8928,100,100,5)

```
import math
import numpy as np

def cor_map(data_set):
    size = int(math.sqrt(num_districts))
```

```
    res = np.zeros((num_timePoints,size,size,num_business),dtype
    for i in range(num_timePoints):
        for j in range(num_districts):
            x = j//size
            y = j%size
            for k in range(num_business):
                res[i,x,y,k] = data_set[i,j,k]

    return res

data = cor_map(data)
```

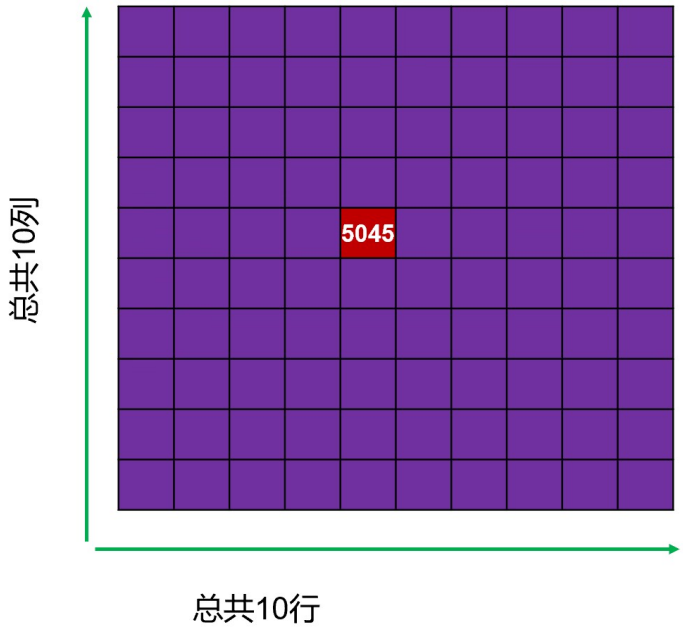
2.3. 划定预测区域

由于我的机器的**硬件上限**(内存最多8G)，而以长度为4的滑动窗口加载数据会直接在内存中塞9个G的数据，内存会爆掉。所以我只能改成选定某个区域，进行预测。

不过理论上我这个代码是具有可扩展性的，在硬件上限较好的机器上，完全可以选定完整的100*100的区域进行训练和预测

这里我以5045小区为中心，选定周围10*10个共100个小区同时进行预测。

由于python的下标从0开始，因此这个范围变成了[45:54],[40:49]。(5045小区本身的索引是(50,44)，意即前面有0:49共50行的50*100=5000个小区，它在第51行的第44个，前面有0:43共44个小区)



说明

内容一第四个小问题指的是：计算这10*10个小区的流量（包含5045）与5045小区流量的皮尔逊相关系数

```
def select_region(sx,ex,sy,ey,data_set):
    m = ex-sx+1
    n = ey-sy+1
    res = np.zeros((num_timePoints,m,n,num_business),dtype=np.fl
    # print(res.shape)
    for i in range(num_timePoints):
        for j in range(m):
            for k in range(n):
                for l in range(num_business):
                    res[i,j,k,l] = data_set[i,ex+j,ey+k,l]
    return res

# 以5045小区为中心
start_x = 46
end_x = 55
start_y = 40
end_y = 49
```

2.4. 划分数据集

将数据集按7:3的比率划分为训练集和测试集。这一块比较简单。

```
seperator_index = int(np.round((num_timePoints*0.7)))
#print(seperator_index)

train_set = data[0:seperator_index,:,:]
test_set = data[seperator_index:num_timePoints,:,:]
print(train_set.shape,test_set.shape)
```

2.5. 构造输入输出序列

CNN希望接收的参数格式: (batch_size, time_steps, features)

放到这个例子中就是(样本数量, 时间步长, 特征数量)。假设我们的时间序列的滑动窗口大小为4,构建输入输出序列

也就是最终的输入序列的形状是(6246,4,10,10,5)，其中batch_size = 6246，time_steps = 4，features = (10,10,5)的一个张量。其中第一个维度代表小区行索引，第二个维度代表小区列索引，第三个维度代表流量业务类型数。

```
def restructure(set, win_size):
    x_seq,y_seq = [],[]
    shape = set.shape
    tps = shape[0]

    for i in range(win_size,tps):
        x_seq.append(set[i-win_size:i,:,:,:])
        y_seq.append(set[i,:,:,:])

    x_seq = np.array(x_seq)
    y_seq = np.array(y_seq)
    return x_seq,y_seq
```

注意滑动窗口构造输入输出序列时，ti:ti+window_size对应的监督为ti+window_size+1的流量。

现在明确一下：

1. X=(train_tps-window_size,window_size,size,size,train_business)
2. Y=(train_tps-window_size,size,size,train_business)

对于具体取值，这里为了更加直观的展现，我还打印了一下：

```
print(X_train.shape,Y_train.shape)
print(X_test.shape,Y_test.shape)
print(train_set.shape,test_set.shape)
```

Executed at 2024.04.30 11:18:17 in 150ms

```
(6246, 4, 10, 10, 5) (6246, 10, 10, 5)
(2674, 4, 10, 10, 5) (2674, 10, 10, 5)
(6250, 10, 10, 5) (2678, 10, 10, 5)
```

2.6. 保存训练集与测试集

如此划分并重塑好的数据集，可以保存下来，以后对于不同的网络模型，可以复用。

```
def save_data(file_path:str,file_name:str):
    with h5py.File(os.path.join(file_path,file_name),'w') as fil
        file.create_dataset('X_train',data=X_train)
        file.create_dataset('Y_train',data=Y_train)
        file.create_dataset('X_test',data=X_test)
        file.create_dataset('Y_test',data=Y_test)
```

并且以后想要加载一个训练好的模型做预测时，我们可以直接从数据集中读出'X_test'和'Y_test'，将predictions和Ground-Truth进行比对。

3. 模型构建与训练

3.1. 构建CNN网络模型

最终我采用的方案是：双卷积+双池化+展平层+全连接层+输出层+重塑层。

在代码上体现为：

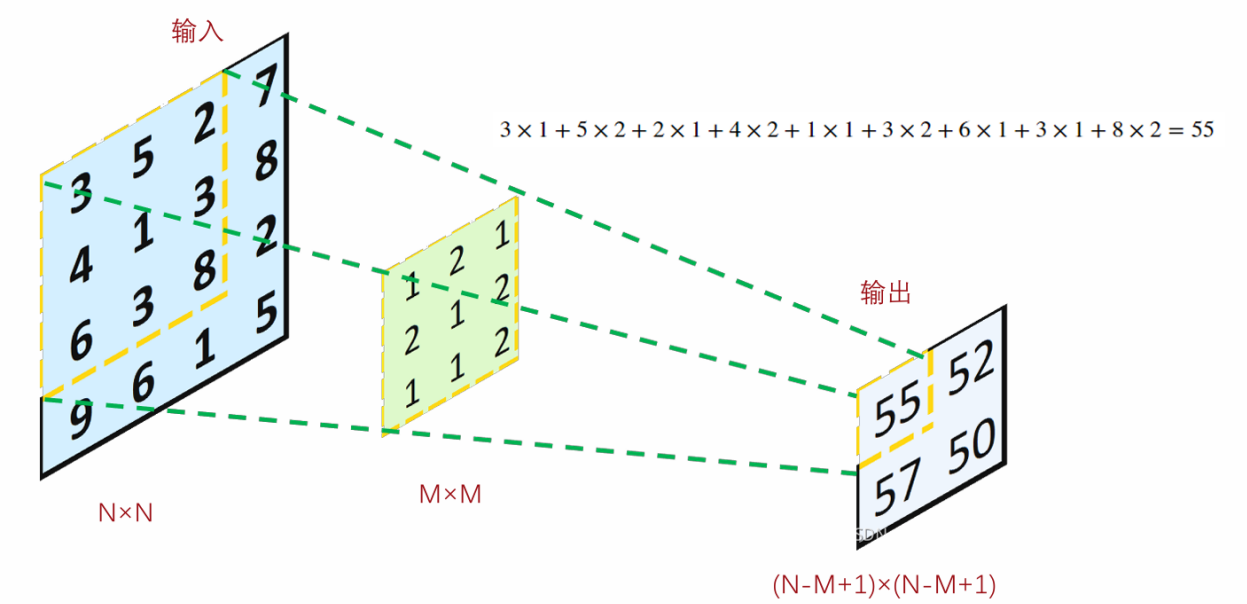
```
input_shape = (window_size,len_x,len_y,num_business)

model = tf.keras.Sequential([
    # 第一个卷积层，使用32个3D卷积核，大小为3x3x3
    Conv3D(32, kernel_size=(3, 3, 3), activation='relu', input_s
    # 池化层，使用2x2x2的池化核，但可能需要调整以避免尺寸问题
    MaxPooling3D(pool_size=(2, 2, 2)),
    # 第二个卷积层，使用64个3D卷积核
    Conv3D(64, kernel_size=(3, 3, 3), activation='relu', padding
    # 另一个池化层，可能需要调整以避免尺寸问题
    MaxPooling3D(pool_size=(2, 2, 2)),
    # 将3D卷积层的输出展平为1D
    Flatten(),
    # 一个全连接层
    Dense(128, activation='relu'),
    # 输出层
    Dense(len_x*len_y*num_business),
    tf.keras.layers.Reshape((len_x,len_y,num_business))
])
```

卷积层

卷积层(Convolution layer)是CNN的核心，它负责提取输入数据的局部特征。每个卷积层都由多个卷积核（也称为滤波器）组成，每个卷积核都可以捕捉到输入数据的一种特定特征。例如，一个卷积核检测边缘，另一个卷积核检测纹理。

举个例子，对于一张4*4的图像，经过3*3的卷积核之后，将剩下2*2的矩阵。

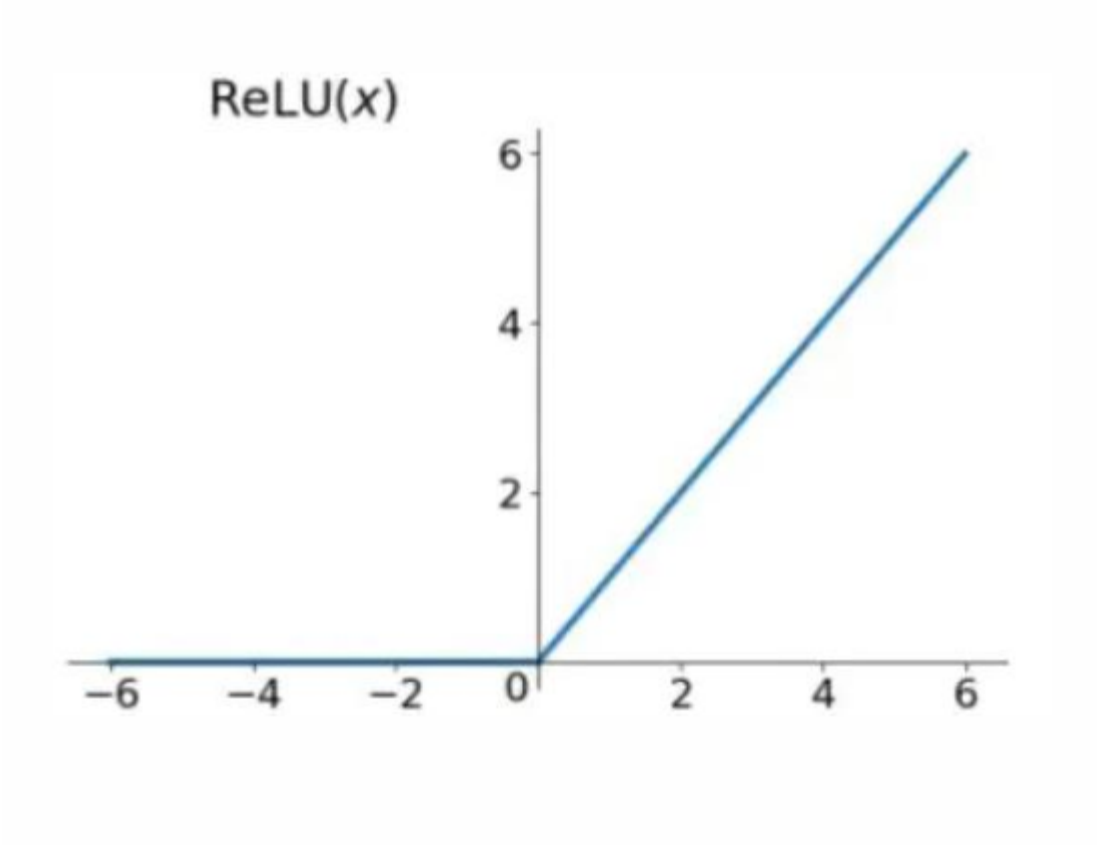


RELU激活函数

激活函数的作用是将卷积的结果映射到一个非线性的空间，这使得模型可以学习到更复杂的特征。

例如ReLU激活函数，

$$y = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{if } x \leq 0. \end{cases}$$



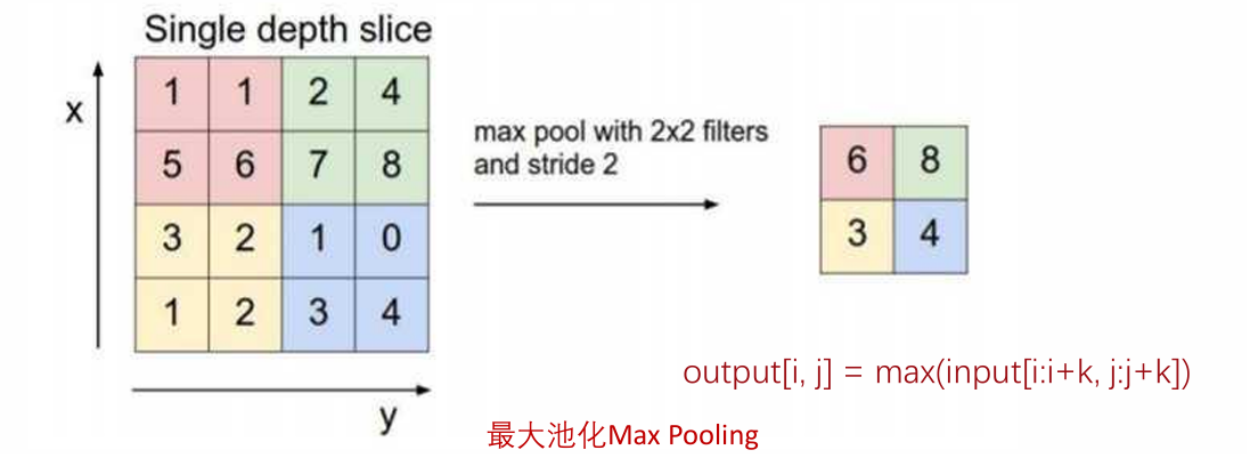
如果正向传播时的输入x大于0，则反向传播会将上游的值原封不动地传给下游。反过来，如果正向传播时的x小于等于0，则反向传播中传给下游的信号将停在此处。

池化层

池化层(pooling layer) 将输入数据划分为多个小的区域，然后在每个区域上计算一个统计值，如最大值或者平均值。这个过程可以看作是对输入数据进行降维，同时保留最重要的信息，从而提高计算效率和模型泛化能力。

例如最大池化（这里用的），将每2*2*2的区块中的最大值提取出来予以保留。

下面是一个二维图像池化的例子，可以看到不同颜色对应的最大值均被提取出来，保留了边缘信息。



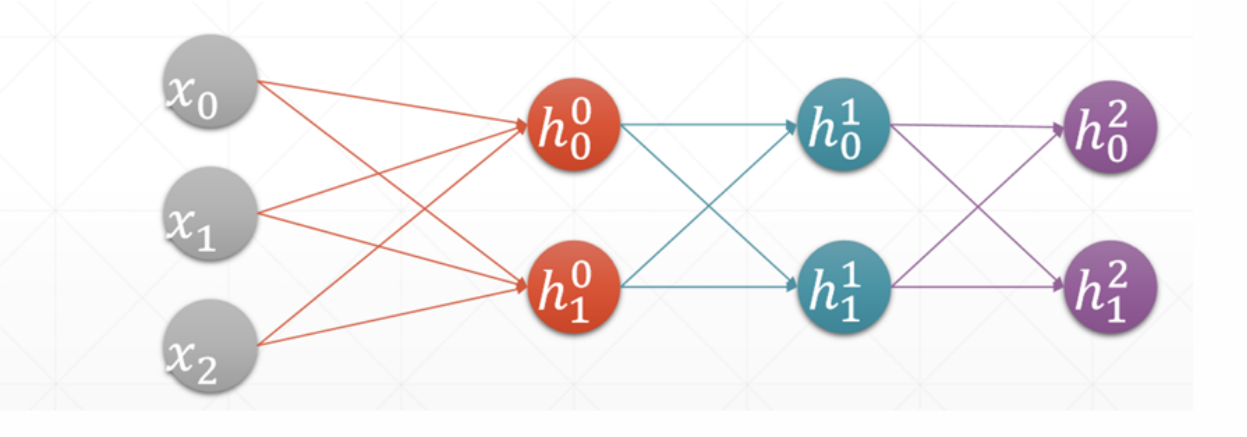
展平层

通常在CNN中，全连接层之前会加入一个**展平层（Flatten Layer）**。展平层的作用是将之前卷积层或池化层输出的多维特征图转换为一维向量，以便连接到全连接层。

展平层将保持特征的顺序，但将它们转换为一维形式，这样全连接层就可以接受它们作为输入。

全连接层

全连接层(Fully Connected Layer)把卷积层和池化层提取出来的所有局部特征重新通过权值矩阵组装成一个完整的图，因为用到了所有的局部特征，所以叫全连接.
全连接层作用是将输入特征映射到输出结果，通常在神经网络的最后一层使用，用于分类、回归等任务。全连接层的输出结果可以看作是对输入特征的一种非线性变换，这种变换可以将输入特征空间映射到输出结果空间，从而实现模型的复杂性和非线性拟合能力



在输出后，我们需要将输出的多维向量重塑为原始的输入维度。也即重塑层。

通过model.summary()，看到模型结构如下：

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv3d (Conv3D)             (None, 4, 10, 10, 32)     4352

 max_pooling3d (MaxPooling3  (None, 2, 5, 5, 32)       0
 D)

 conv3d_1 (Conv3D)           (None, 2, 5, 5, 64)       55360

 max_pooling3d_1 (MaxPoolin  (None, 1, 2, 2, 64)       0
 g3D)

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 128)               32896

 dense_1 (Dense)             (None, 500)               64500

 reshape (Reshape)           (None, 10, 10, 5)         0

=================================================================
Total params: 157108 (613.70 KB)
Trainable params: 157108 (613.70 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

3.2. 训练模型

加入早停机制

为了防止模型过拟合，我们在epoch无法改善loss时进行早停：

```
from tensorflow.keras.callbacks import EarlyStopping

# 初始化 EarlyStopping 回调
early_stopping = EarlyStopping(
    monitor='val_loss',  # 监控的指标
    patience=3,          # 允许的没有改善的 epoch 数量
    verbose=1,           # 详细模式，输出早停信息
    restore_best_weights=True  # 恢复到最佳的模型权重
)
```

训练时，加入早停机制

```
history = model.fit(X_train, Y_train, epochs=100, batch_size=32,
                validation_split=0.2, verbose=1,callbacks=[early
```

解释下几个参数：

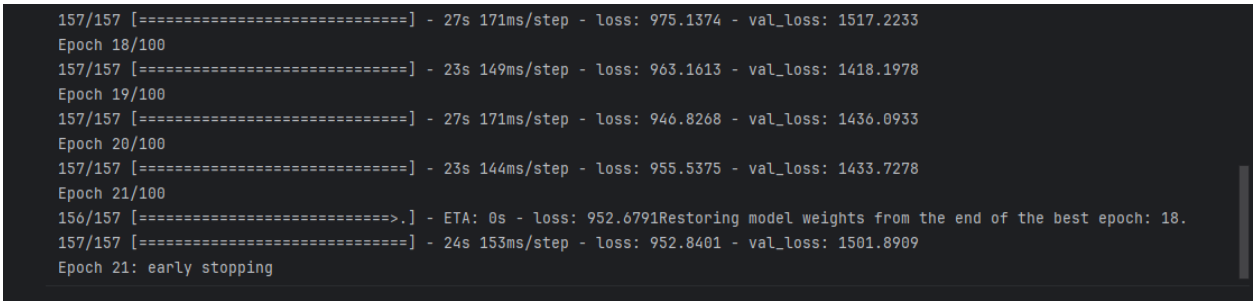
1. epochs=100，10轮训练
2. batch_size=32，每轮训练每次接受32个输入输出进行训练
3. validation_split：以训练集的20%作为验证
4. verbose=1，提供训练的详细输出
5. calbacks，训练时的回调函数，这里我们加入了早停机制的回调。

```
Epoch 1/10
157/157 [==============================] - 33s 183ms/step - loss
Epoch 2/10
157/157 [==============================] - 30s 189ms/step - loss
Epoch 3/10
157/157 [==============================] - 29s 183ms/step - loss
Epoch 4/10
157/157 [==============================] - 32s 202ms/step - loss
Epoch 5/10
157/157 [==============================] - 30s 188ms/step - loss
Epoch 6/10
```

```
157/157 [==============================] - 30s 191ms/step - loss
Epoch 7/10
157/157 [==============================] - 28s 180ms/step - loss
Epoch 8/10
157/157 [==============================] - 24s 151ms/step - loss
Epoch 9/10
157/157 [==============================] - 26s 167ms/step - loss
Epoch 10/10
157/157 [==============================] - 25s 162ms/step - loss
... # 太多了后面不放了
```

可以看到每一轮输出的用时、损失都打印了出来。

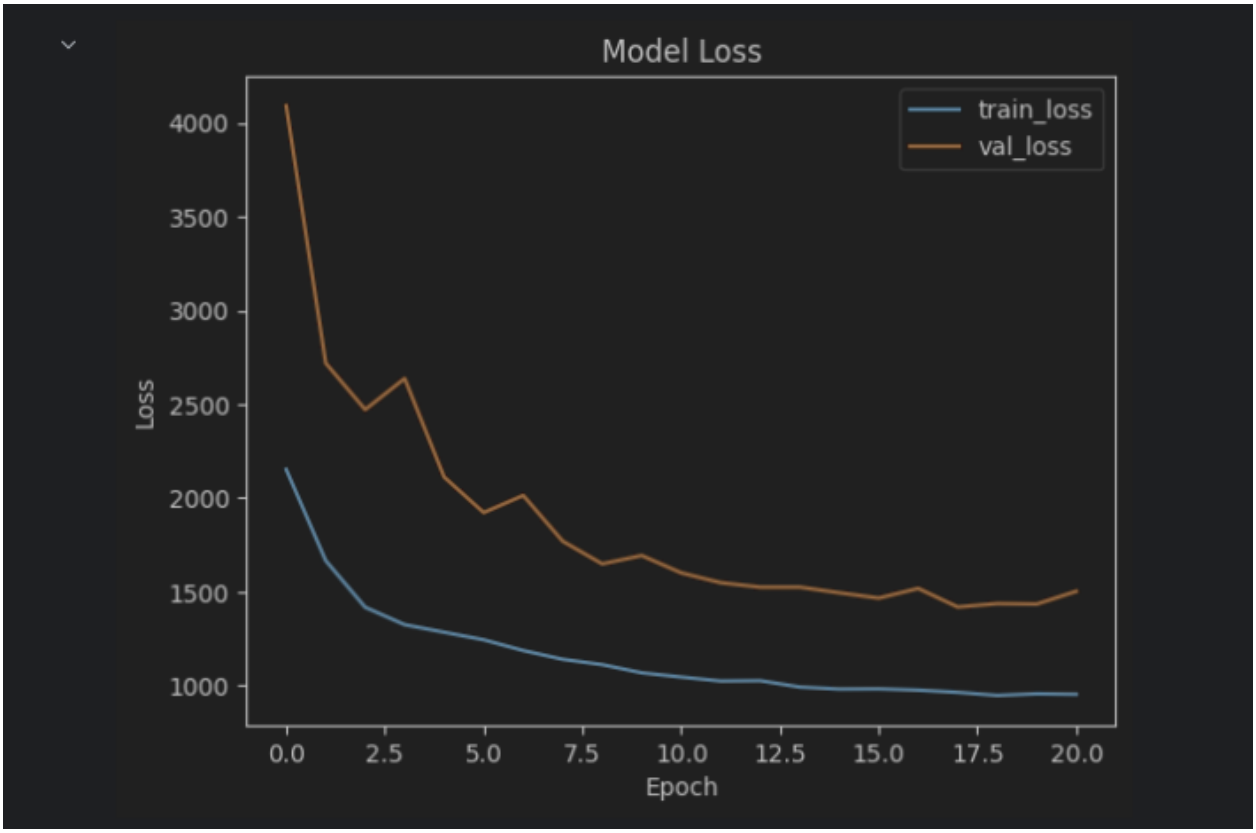
而在21轮时进行了早停：



绘制epoch-loss图像

我们可以绘制epoch-loss图像，查看随着训练轮数的增加，loss的改变：

```
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')
plt.show()
```



3.3. 保存模型

训练好的模型可以保存起来，日后复用。

```
# 构建模型文件名
model_filename = f'model_{start_x_str}_{end_x_str}_{start_y_str}

model_path = paths['model_path']
model.save(os.path.join(model_path,model_filename))
```

3.4. 评估性能

我们可以采用测试集来评估这个模型的性能，例如

```
# 评估模型
loss = model.evaluate(X_test, Y_test, verbose=1)
print(f"Test loss: {loss}")
```

可以看到损失大概是1400+。这里损失这么大，是因为我没有做数据的标准化，因为本身量纲就是统一的，所以没必要去量纲。

3.5. 模型预测与预测值分析

简单的几个指标：

MAE（平均绝对误差）：

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

MSE(均方误差）：

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

RMSE（均方根误差）：

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

```
predictions = model.predict(X_test)
```

```
assert predictions.shape == Y_test.shape, "确保 predictions 和 Y_

# 计算MSE
mse = np.mean((predictions - Y_test) ** 2)

# 计算RMSE
rmse = np.sqrt(mse)

# 计算MAE
mae = np.mean(np.abs(predictions - Y_test))

print(f'MAE: {mae}')
print(f'MSE: {mse}')
print(f'RMSE: {rmse}')
```

得到：

```
MAE: 16.309743063784293
MSE: 1361.798195288669
RMSE: 36.90254998355356
```

对于预测值的进一步分析，我将写在实验四的文档中。