第5章

基于深度学习的共享单车轨迹数据案例实战

5.1 研究背景

近年来，随着移动互联网技术的发展和基于位置服务的普及，大量的传感器被布置在城市的各个角落，以实时感知和记录人在城市中的位置和时间信息，如 GPS 全球定位系统、手机基站等。这些被记录下来的具有时空信息的各种数据被统称为城市时空大数据，包括人的轨迹数据、共享单车数据和出租车数据等。共享单车数据是城市时空大数据的重要组成部分，通过对共享单车数据进行深入的分析与挖掘，研究人员可以发现有价值的知识，从而帮助政府和城市管理者进行更合理的城市规划与管理，提升城市运行效率，实现城市的可持续发展。

城市共享单车出入流预测是共享单车数据挖掘的重要研究内容，通过对本问题的研究，可以从一定程度上反映城市居民的出行规律，缓解共享单车在不同区域供需不平衡的问题，提前发现未来某一时段内共享单车需求量暴增或者共享单车数量不足的问题，以做好单车调度、单车分配等任务，缓解城市中的“最后一千米问题”，极大地方便个人生活。同时，对不同城市区域居民的出行习惯和出行规律进行分析，辅助城市管理者进行城市规划。

本章利用深度学习和迁移学习方法，以基于深度域适应网络的共享单车出入流知识迁移与预测为应用背景，对现有的公开共享单车数据进行深入分析，学习共享单车复杂的时空分布，进行知识迁移，并进行预测。以“数据获取—数据预处理—应用实战”为主线，带领初学者完整实现一套标准的深度学习建模流程。

5.2 研究现状

共享单车正在给城市的交通结构带来巨大的变革，自 2016 年以来，中国市场上涌现

出大量的共享单车产品，如摩拜单车、ofo 小黄车和小蓝单车等。目前，我国共享单车市场有着庞大的市场需求，利用数据挖掘技术对共享单车的骑行规律进行建模，有巨大的应用前景和社会意义。本章主要讨论共享单车数据挖掘的两个主要研究方向，其一是共享单车出入流预测，即利用城市中不同区域的历史共享单车数据预测未来一段时间内的共享单车出入流数据，现有的出入流预测方法一般通过深度学习等技术构建模型，并使用天气等外部数据辅助预测。其二是共享单车调度优化研究，例如，采用传统的遗传算法、蚁群算法等对单车调度路径优化、引入新兴的强化学习技术对共享单车的调度路线进行优化等。下面对两个研究方向进行详细简介。

5.2.1　共享单车出入流预测研究

共享单车出入流预测与前述人工智能在轨道交通运营管理的主流研究方向章节中（第 4 章）城市轨道交通短时进站流预测部分类似，故本节只做简要概述。

如果仅关注某一特定位置在一段时间内的相关共享单车测量数据，那么可以将其建模成时间序列数据，如某条道路上的共享单车流量数据。在传统的研究上，研究者们大都使用基于统计的方法，如 ARIMA、SVR 与贝叶斯网络等。但是这些基于统计的方法学习到的特征比较浅，由于它们有限的学习能力，无法学习到复杂的时空依赖。

随着深度学习技术的进步，深度学习技术被广泛应用到共享单车出入流预测上。一类方法通过将整个城市看作图像，并应用卷积神经网络（CNN）捕获空间特征，进行区域级的共享单车出入流预测。由于时间特征对共享单车预测研究十分重要，又引入循环神经网络（RNN），并结合 CNN 同时学习时空特征，引入了如 ConvLSTM 等模型，并进一步扩展出 SeqST-GAN 等方法。但是，随着研究的深入，研究学者们发现目前的算法将整个城市看作一张图片，然后使用卷积神经网络（CNN）对其进行处理，但是这种方式不能完整地反映空间关系。例如，城市中的两个区域相隔比较远，但是它们却有着相似的空间模式，而传统的卷积神经网络（CNN）无法捕获这种空间特征。为了捕获这种全局的空间特征，对这种非欧式数据进行建模，研究者们又提出了另一类研究方法，即基于网神经网络（GNN）的共享单车出入流预测方法，通过引入随机游走（Random Walk）、图卷积神经网络（GCN）等方法，并结合 RNN 与最近很火的 Transformer 等方法，提出了很多方法，如 AEST 等。同时，也可以将图像的每一个区域看作节点，将 DCRNN、STGCN 和 GMAN 等方法应用于共享单车出入流预测。

随着数据种类的增加，很多研究者引入了迁移学习、联合训练模型和多任务学习模型对共享单车的出入流预测进行了研究。来自北京大学等单位的研究者们提出了名为 RegionTrans 的迁移学习方法，首先利用现有数据计算目标域中的每个区域和源域直接的相似性，然后将这些相似性加入到基于深度学习的预测模型中作为约束，将数据丰富的城市数据知识迁移到数据稀疏的城市中。笔者提出了一种统一的端到端的时空域适应网络 ST-DAAN，从数据分布的角度，借助深度域适应网络 DAN 与最大均值差异 MMD，将丰富的源域知识迁移到目标域中，以辅助数据稀疏的城市进行共享单车的出入流预测。同时，为了考虑出入流预测与起点-目的地（OD）预测的相关性，笔者又提出了 MT-ASTN 模型，利用基于对抗学习的多任务学习方法，学习两个任务的公有特征，同时辅助共享单

车出入流预测与 OD 预测。

5.2.2 共享单车调度优化研究

共享单车的调度优化研究能为城市管理者与共享单车运营商管理共享单车提供参考与理论支撑，其受多维因素影响，如天气、节假日等。现对国内外共享单车调度优化研究进行简要综述。

国外对共享单车的调度优化研究比较早，在早期的研究中，研究者们主要使用基于规则的算法进行共享单车的调度优化。研究者们一开始通过建立总调度距离最短模型和分支切割算法对调度车辆唯一情况下的调度问题进行求解；紧接着，另一部分研究者研究了共享单车调度路线的确定、迁移或自行车放置路线问题，通过将目标函数定义为调度总距离及被租赁点拒绝的使用者数量进行优化；随后，有研究者引入数据挖掘方法，利用聚类分析，对单车的租赁点进行聚类，对不同租赁点之间的线路分布进行建模；同时，又有研究者引入系统动力学模型解决共享单车系统的动态平衡问题，求解不同状态下系统的最佳调度方案；考虑到调度成本问题，研究者们提出了基于调度成本最小化的单车二次配送的动态模型，模型得出了客户满意度最高的最优调度路径；为了对不同站点不同车辆进行统一规划，提出了混合整数规划模型，为大规模公共自行车系统的调度和再分配问题提供了一种新思路；还有一部分研究者们借助机器学习方法，将能考虑到的影响共享单车调度因素作为特征输入到机器学习模型中，优化共享单车调度；随着强化学习的火爆，现在的研究者们引入强化学习对共享单车的调度路线进行优化，以减少成本。但是国外对调度问题的研究主要基于有桩共享单车，涉及无桩共享单车的调度研究较少。

国内的学者对共享单车的调度优化也有很多重要的研究。起初，研究者们利用遗传算法和禁忌搜索法，通过将运营成本与服务质量作为约束，对需求不断变化的供需单车调度问题进行求解；后来，有研究者发现共享单车的供需关系在时空上都存在不均衡问题，使用蚁群算法对单车调度路径优化问题进行求解；还有研究者通过归纳分析借还车数据，建立 BP 神经网络预测借还车的分布情况，通过定义调度时间窗内站点的饱和度分析最优调度方案；另一部分研究者利用聚类等数据挖掘方法分析居民使用共享单车的出行规律，对时空特征建模，为共享单车调度提供参考。

5.3 数据获取手段及开源数据集简介

共享单车领域的数据集主要包含纽约有桩共享单车数据集、芝加哥共享单车数据集、摩拜单车数据集等。本章的案例主要以基于经纬度的共享单车的出入流预测为主，因此，此处会着重对纽约有桩共享单车数据集和芝加哥共享单车数据集的获取手段以及相关的数据集信息进行简介，同时也会对摩拜单车数据集进行简要介绍。

1. 纽约有桩共享单车数据集 CitiBike

CitiBike 是纽约有桩共享单车的轨迹数据，包含 2013 年 6 月～2021 年 7 月（截至笔者撰写本节前）全纽约市范围内城市有桩共享单车的数据，本节以 2015 年 1 月～

2015 年 12 的数据为例，总的来说，CitiBike 在纽约建立了超过 600 个自行车站点，并投放了 10 000 辆左右的自行车，数据集中的每条轨迹数据包含 11 个字段：行程时间、行程开始日期/时间、行程结束日期/时间、起始站点编号、起始站点名称、起始站点经度、起始站点纬度、终止站点编号、终止站点名称、终止站点经度、终止站点纬度、自行车编号、用户类型、生日、性别。官网目前可直接下载 2015 年 1 月 1 日～2015 年 12 月 31 日的数据，该数据集超过九百万条。CitiBike 官网（https://www.citibikenyc.com/system-data）提供了其他年份的全量数据可供下载，且提供了详细的数据集说明。本节以热力图的形式可视化了整个纽约共享单车数据集 2015 年某个时刻的签入签出数据。CitiBike 签出可视化如图 5-1 所示。

CitiBike 签入可视化如图 5-2 所示。

图 5-1　CitiBike 签出可视化

图 5-2　CitiBike 签入可视化

2. 芝加哥共享单车数据集 DivvyBike

DivvyBike 是基于经纬度的芝加哥公开共享单车数据集，由芝加哥自行车共享系统 Divvy 进行数据采集，包含 2013 年 1 月～2021 年 7 月全芝加哥市范围内城市共享单车的数据，本节以 2015 年的共享单车数据为例。总的来说，DivvyBike 建立了 580 个站点，并投放了 5800 辆自行车。从 2015 年 1 月至 2015 年 12 月，DivvyBike 拥有超过 6 百万条自行车行程数据，其中每条行程数据包含以下字段：行程开始时间、行程结束时间、起始站点编号、起始站点名称、起始站点经度、起始站点纬度、终止站点编号、终止站点名称、终止站点经度、终止站点纬度、自行车编号、用户类型、生日和性别。用户可通过 Divvy Data 官网（https://www.divvybikes.com/system-data）下载全量数据，同时官网也提供相应的数据说明和数据使用条例。本节以热力图的形式可视化了整个芝加哥共享单车数据集 2015 年某个时刻的签入签出数据。DivvyBike 签出可视化如图 5-3 所示。

DivvyBike 签入可视化如图 5-4 所示。

图 5-3 DivvyBike 签出可视化

图 5-4 DivvyBike 签入可视化

3. 摩拜单车数据集 MoBike

MoBike 北京摩拜单车数据集来源于 2017 年中国人工智能学会与摩拜联合举办的 2017 摩拜杯算法挑战赛，大赛以摩拜单车推出以来，已经在很多城市成为除公共交通以外的居民出行方式的首选为背景，主要目标是利用机器学习等技术预测用户的出行目的地等。MoBike 数据集没有固定的站点，在北京投放了超过 40 万辆共享单车，其中每条数据包含 7 个字段：订单编号、用户编号、车辆编号、车辆类型、骑行起始时间、骑行起始区块位置、骑行目的地区块位置。地理位置信息，如骑行起始区块位置和骑行目的地区块位置信息，通过 Geohash 加密，可以通过开源的方法获得详细的经纬度数据。

完整的数据集可通过本书前言提供的获取方式进行获取。

5.4 数据预处理及可视化

本节所使用的代码主要基于经纬度的共享单车数据集，如 CitiBike、DivvyBike 等。首先，对数据集进行简单的分析及预处理，并进行可视化。然后，进一步讲述如何根据经纬度，对整个城市进行网格划分，将整个城市看作一张“图片”，从原始 GPS 轨迹数据提取城市不同区域的出入流时空信息。

以 2015 年 1 月 CitiBike 数据为例，首先利用 Python 3 和 Pandas 对数据进行读取，并输出前 5 条示例数据，代码如下。

```
import pandas as pd
df = pd.read_csv('./Dataset/NYC_Citybike/2015/201501-citibike-tripdata.csv', sep=',')
print(df.head())
```

2015 年 1 月 CitiBike 示例如图 5-5 所示。

	tripduration	starttime	stoptime	start station id	start station name	start station latitude	start station longitude	end station id	end station name	end station latitude	end station longitude	bikeid	usertype	birth year	gender
0	1346	1/1/2015 0:01	1/1/2015 0:24	455	1 Ave & E 44 St	40.750020	-73.969053	265	Stanton St & Chrystie St	40.722293	-73.991475	18660	Subscriber	1960.0	2
1	363	1/1/2015 0:02	1/1/2015 0:08	434	9 Ave & W 18 St	40.743174	-74.003664	482	W 15 St & 7 Ave	40.739355	-73.999318	16085	Subscriber	1963.0	1
2	346	1/1/2015 0:04	1/1/2015 0:10	491	E 24 St & Park Ave S	40.740964	-73.986022	505	6 Ave & W 33 St	40.749013	-73.988484	20845	Subscriber	1974.0	1
3	182	1/1/2015 0:04	1/1/2015 0:07	384	Fulton St & Waverly Ave	40.683178	-73.965964	399	Lafayette Ave & St James Pl	40.688515	-73.964763	19610	Subscriber	1969.0	1
4	969	1/1/2015 0:05	1/1/2015 0:21	474	5 Ave & E 29 St	40.745168	-73.986831	432	E 7 St & Avenue A	40.726218	-73.983799	20197	Subscriber	1977.0	1

图 5-5　2015 年 1 月 CitiBike 示例

接着，由于字段太多，可以稍加处理只取所需信息字段，并更改字段名，使其更为直观，代码如下。

```
df_columns = [
    'starttime', 'stoptime',
    'start station longitude', 'start station latitude',
    'end station longitude', 'end station latitude'
]
new_col_name = [
    'pick_up_time', 'drop_off_time',
    'pick_up_lon', 'pick_up_lat',
    'drop_off_lon', 'drop_off_lat'
]
df = df.loc[:, df_columns]
df.columns = new_col_name
print(df.head())
```

所需字段信息示例如图 5-6 所示。

	pick_up_time	drop_off_time	pick_up_lon	pick_up_lat	drop_off_lon	drop_off_lat
0	1/1/2015 0:01	1/1/2015 0:24	-73.969053	40.750020	-73.991475	40.722293
1	1/1/2015 0:02	1/1/2015 0:08	-74.003664	40.743174	-73.999318	40.739355
2	1/1/2015 0:04	1/1/2015 0:10	-73.986022	40.740964	-73.988484	40.749013
3	1/1/2015 0:04	1/1/2015 0:07	-73.965964	40.683178	-73.964763	40.688515
4	1/1/2015 0:05	1/1/2015 0:21	-73.986831	40.745168	-73.983799	40.726218

图 5-6　所需字段信息示例

然后利用 Matplotlib 包，统计并可视化共享单车于不同经纬度的使用频率，本节以乘客的上车点为例，经纬度间隔设置为 0.002 代码如下。

```
import matplotlib.pyplot as plt
plt.hist(df['pick_up_lat'], bins = list(np.arange(40.67,40.78,0.002)))
plt.title('Latitude Hist Map')
plt.plot()
plt.hist(df['pick_up_lon'], bins = list(np.arange( - 74.02, - 73.95,0.002)))
plt.title('Pick Up Longitude Hist Map')
plt.plot()
```

共享单车上车点于不同经度使用频率分布如图 5-7 所示。

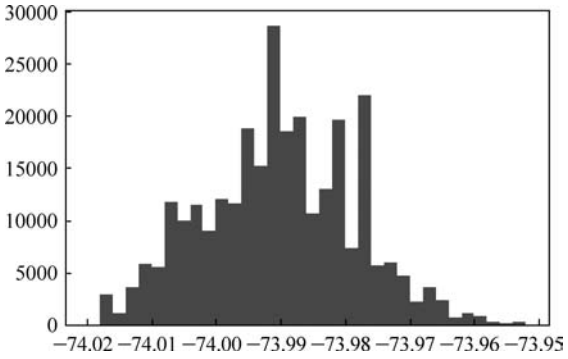

图 5-7　共享单车上车点于不同经度使用频率分布

共享单车上车点于不同纬度使用频率分布如图 5-8 所示。

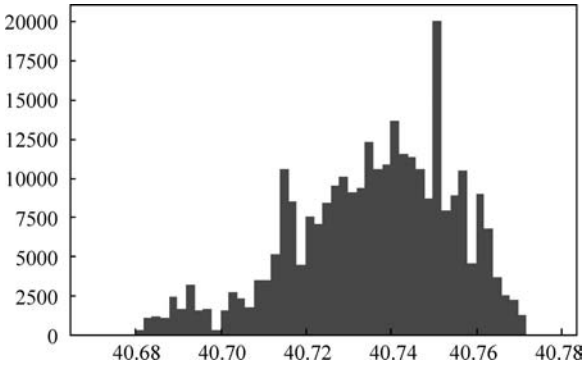

图 5-8　共享单车上车点于不同纬度使用频率分布

分析图 5-7 和图 5-8 可以发现，人们在市中心的活动频率比较高，符合真实世界的规律。

最后，取样一部分数据，分析数据的空间分布，并利用散点图进行可视化，代码如下。

```
sample_index = random.sample(list(range(len(df_main))),int(len(df_main)/30))
df_sample = df.iloc[sample_index,:]
plt.figure(figsize = (10,10))   # 画图的大小
plt.scatter(df_sample['pick_up_lon'], df_sample['pick_up_lat'],alpha = 0.3,s = 0.2)
plt.plot()
```

2015 年 1 月 CitiBike 采样数据空间分布(横轴：经度，纵轴：纬度)如图 5-9 所示。结合纽约市地图，可以发现，大部分使用共享单车的乘客集中在纽约市中心，符合客观规律。本节只给出简单可视化，若读者有兴趣将结果可视化于地图上以获取更清晰的效果，可参考 OpenStreetMap 操作手册。

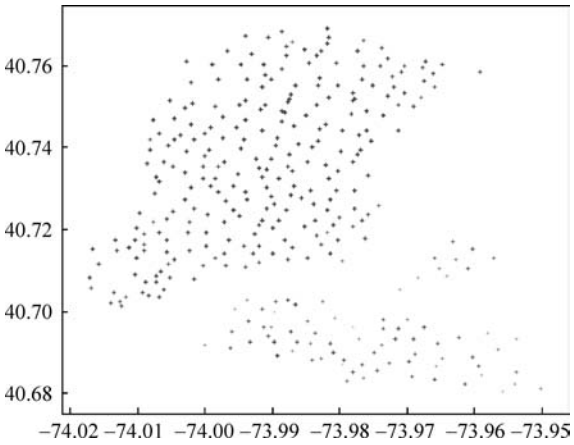

图 5-9　2015 年 1 月 CitiBike 采样数据空间分布(横轴：经度，纵轴：纬度)

通过上述分析，能够大致了解共享单车数据的时空分布与特性。本章的主要研究对象是共享单车的出入流数据，首先，将整个城市看作一张“图像”，对其进行网格划分，每个网格可以类比于图像中的像素，然后将每个网格的出流、入流看作“图像”的通道。在正式讲述代码前，先给出几个必要的定义来帮助读者更好地理解上述概念。

定义 1（子区域）：基于经纬度，将城市分成 $m \times n$ 个网格，每个网格为一个子区域且大小相等，用 $R = r_{1,1}, \cdots, r_{i,j}, \cdots, r_{\{m,n\}}$ 来表示这些子区域，其中，$r_{i,j}$ 代表第 i 行、第 j 列的子区域。

定义 2(出入流“图像”)：设$\mathcal{P}$是共享单车轨迹的集合，给定一个子区域 $r_{i,j}$，其相应的入流与出流定义如下：

$$x^t_{\text{in},i,j} = \sum_{T_r \in \mathcal{P}} |l > 1 \mid g_{l-1} \notin r_{i,j} \wedge g_l \in r_{i,j}|$$

$$x^t_{\text{out},i,j} = \sum_{T_r \in \mathcal{P}} |l > 1 \mid g_l \in r_{i,j} \wedge g_{l+1} \notin r_{i,j}|$$

其中，$T_r : g_1 \rightarrow g_2 \rightarrow \cdots \rightarrow g_{T_r}$ 是$\mathcal{P}$中在时间间隔 t 的一段子轨迹，$g_l \in r_{i,j}$ 表示 g_l 位于区域 $r_{i,j}$ 中，$|\cdot|$代表集合的势。将在时间间隔 t 的入流和出流定义为人流张量(Tensor) $\mathcal{X}^t \in \mathcal{R}^{m \times n \times 2}$，其中，$\mathcal{X}^t_{i,j,0} = x^t_{\text{in},i,j}$，$\mathcal{X}^t_{i,j,1} = x^t_{\text{out},i,j}$。

现从代码角度讲述如何将基于 GPS 的原始轨迹数据进行区域划分并建模成上述出入流“图像”数据。

首先，导入所需要的包，定义函数 geo_to_grid()，将城市划分成网格，geo_to_grid()函数返回子区域的编号，代码如下。

```
import numpy as np
import pandas as pd
```

```
import os
import multiprocessing as mul
import time


def geo_to_grid(geo):
    lat, lng = geo[0], geo[1]
    if (lat > LAT_STOP
            or lat < LAT_START
            or lng > LNG_STOP
            or lng < LNG_START):
        return -1
    lng_ind = int(np.floor((lng - LNG_START) / LNG_INTERVAL))
    lat_ind = int(np.floor((lat - LAT_START) / LAT_INTERVAL))
    return lng_ind, lat_ind
```

然后，定义 bike_get_day_hour()函数，对原始轨迹中的时间字段进行解析，获取当前记录的日期和时间，代码如下。

```
def bike_get_day_hour(x):
    time_list = x.split(' ')
    date_str = time_list[0]
    date_list = date_str.split('/')
    time_str = time_list[1]
    day = int(date_list[1])
    hour = int(time_str.split(':')[0])
    return day, hour
```

结合上述数据分析与预先定义的 geo_to_grid()与 bike_get_day_hour()函数，对原始轨迹数据进行解析，以 1 小时为时间间隔，将城市划分成子区域，并获取数据集中不同日期的不同时间段各个子区域的出入流数据，并存储到预先定义好的 flow_array 矩阵中，最后将处理好的数据按月存储到本地当前目录下，代码如下。

```
def bike_stat(date_str):
    tl = []
    tl.append(time.time())
    df = pd.read_csv(data_path + '{}-citibike-tripdata.csv'.format(date_str), sep=',')
    tl.append(time.time())
    print('%s Data loaded in %.2fsec' % (date_str, (tl[-1]-tl[-2])))

    df_columns = [
        'starttime', 'stoptime',
        'start station longitude', 'start station latitude',
        'end station longitude', 'end station latitude'
    ]
```

```
    new_col_name = [
        'pick_up_time', 'drop_off_time',
        'pick_up_lon', 'pick_up_lat',
        'drop_off_lon', 'drop_off_lat'
    ]

    df = df.loc[:,
df_columns]
    df.columns = new_col_name
    df_main = df[
        (LNG_START < df.pick_up_lon) & (df.pick_up_lon < LNG_STOP)
        & (LAT_START < df.pick_up_lat) & (df.pick_up_lat < LAT_STOP)
        & (LNG_START < df.drop_off_lon) & (df.drop_off_lon < LNG_STOP)
        & (LAT_START < df.drop_off_lat) & (df.drop_off_lat < LAT_STOP)
        ]

    df_main.pick_up_time = df_main.pick_up_time.apply(str)
    df_main.drop_off_time = df_main.drop_off_time.apply(str)
    with mul.Pool(48) as pool:
        pick_up_array = np.array(list(pool.map(bike_get_day_hour, df_main.pick_up_
time)))
        drop_off_array = np.array(list(pool.map(bike_get_day_hour, df_main.drop_off_
time)))
        start_grid_array = np.array(list(pool.map(geo_to_grid, df_main.loc[:, ["pick_up
_lat", "pick_up_lon"]].values)))
        end_grid_array = np.array(list(pool.map(geo_to_grid, df_main.loc[:, ["drop_off_
lat", "drop_off_lon"]].values)))
    tl.append(time.time())
    print('Get %s Using %.2f min' % (date_str, (tl[-1]-tl[-2])/60))
    df_main['start_x'] = start_grid_array[:, 0]
    df_main['start_y'] = start_grid_array[:, 1]
    df_main['end_x'] = end_grid_array[:, 0]
    df_main['end_y'] = end_grid_array[:, 1]
    df_main['start_day'] = pick_up_array[:, 0]
    df_main['start_hour'] = pick_up_array[:, 1]
    df_main['end_day'] = drop_off_array[:, 0]
    df_main['end_hour'] = drop_off_array[:, 1]

    start_g = df_main.groupby(['start_day','start_hour', 'start_x', 'start_y'])
    end_g = df_main.groupby(['end_day', 'end_hour', 'end_x', 'end_y'])
    start_count = start_g.agg('count').reset_index()
    end_count = end_g.agg('count').reset_index()
    print(date_str, 'have {} days which contains {} hours'.format(max(df_main.start_day),
max(df_main.start_hour) + 1))
    flow_array = np.zeros((max(df_main.start_day), max(df_main.start_hour) + 1,
INTERVAL_NUM, INTERVAL_NUM, 2))
    for row in start_count.iterrows():
        x_vle = row[1]['start_x']
```

```
            y_vle = row[1]['start_y']
            day = row[1]['start_day'] - 1
            hour = row[1]['start_hour']
            times = row[1]['end_day']
            try:
                flow_array[day, hour, x_vle, y_vle, 0] = times
            except:
                print(row[1])
        for row in end_count.iterrows():
            x_vle = row[1]['end_x']
            y_vle = row[1]['end_y']
            day = row[1]['end_day'] - 1
            hour = row[1]['end_hour']
            times = row[1]['start_day']
            try:
                flow_array[day, hour, x_vle, y_vle, 1] = times
            except:
                print(row[1])
        flow_array.tofile('./ ' + 'bike_ % s' % date_str)
        print('% s Finished % .2f min' % (date_str, (tl[ - 1] - tl[ - 2]) / 60))
        print('*' * 100)
```

上述代码首先读取 CitiBike 原始轨迹数据，提取原始所需信息，如时间、经纬度等，并设置经纬度过滤条件，即所在城市的经纬度范围，将在所需经纬度之外的数据点作为异常点删除，然后对城市进行网格划分，并获取相应轨迹数据的起始点网格编号、终止点网格编号、日期、时间等信息，根据上述出入流公式，计算每个网格代表的子区域的出入流，并存储到矩阵对应的位置，最后将矩阵本地化存储于当前目录，代码如下。

```
def bike_count():
    global INTERVAL_NUM, LNG_START, LNG_STOP, LNG_INTERVAL
    global LAT_START, LAT_STOP, LAT_INTERVAL
    global data_path, bike_path
    data_path = './DataSet/'
    bike_path = './DataSet/'
    #bike lng/lat
    INTERVAL_NUM = 32
    LNG_START = - 74.02
    LNG_STOP = - 73.95
    LNG_INTERVAL = abs(LNG_STOP - LNG_START) / INTERVAL_NUM
    LAT_START = 40.67
    LAT_STOP = 40.77
    LAT_INTERVAL = abs(LAT_STOP - LAT_START) / INTERVAL_NUM
    for year in range(2015, 2016):
        for mon in range(1, 13):
            if mon < 10:
                date_str = str(year) + '0' + str(mon)
            else:
```

```
                date_str = str(year) + str(mon)
            file_str = bike_path + 'bike_%s' % date_str
            if os.path.exists(file_str):
                print('bike_%s' % date_str + 'Exist!!!')
            else:
                bike_stat(date_str)


if __name__ == '__main__':
    bike_count()
```

最后，定义 bike_count()函数，给定经纬度范围、网格数量，假设数据存储于./DataSet/目录中，对 2015 年 1 月～2015 年 12 月的纽约共享单车数据集 CitiBike 进行处理，并按月存储到本地。出入流数据按月本地存储结果示例如图 5-10 所示。

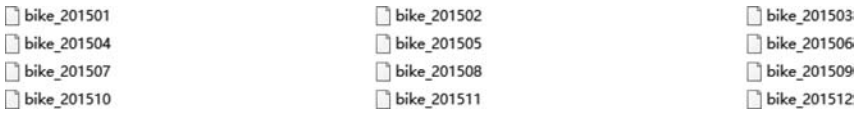

图 5-10　出入流数据按月本地存储结果示例

5.5　基于 PyTorch 的共享单车数据建模

5.5.1　问题陈述及模型框架

本节将基于公开的共享单车 CitiBike 数据集与纽约出租车数据集 NYCTaxi，构建一个简单的基于迁移学习的深度学习模型，进行实战应用与详解。在真实应用中，由于各种原因，如数据采集机制落后、数据隐私保护等，笔者在某些城市能获取到的共享单车数据可能十分有限，而城市中的某些时空数据又十分丰富，如出租车数据等。出租车数据与共享单车数据存在时空相关性，可以借助出租车数据蕴含的知识辅助共享单车进行预测，本节利用迁移学习中深度域适应网络的思想，利用最大均值差异（Maximum Mean Discrepancy，MMD），建立一个深度时空域适应网络，从数据分布的角度，借助数据丰富的源域城市知识辅助数据稀疏的目标域城市，进行共享单车的出入流预测。因深度域适应网络、最大均值差异等方法不是本书的重点，且可以通过各种方式查询到大量资料，故此处不再赘述。

深度时空域适应网络模型框架如图 5-11 所示，给定一个数据充足的源域和一个数据稀疏的目标域，首先采用堆叠的 3D 卷积(Conv3D)网络将原始的时空共享单车数据映射到一个公共嵌入空间中，然后使用全连接网络(FC)学习每个域特有的特征，并将这些特征嵌入到希尔伯特可再生核空间中，利用最大均值差异 MMD 作为约束，减少域间差异，从而达到知识迁移的目的，最终，借助一层全连接网络进行预测。在本节中，假设纽约出租车数据 NYCTaxi 丰富，纽约共享单车 CitiBike 数据稀疏，故将 NYCTaxi 作为源域，CitiBike 作为目标域，具体来说，需要将所有经过数据预处理的 NYCTaxi 训练数据输入

模型，而随机采样一部分 CitiBike 数据用于训练。

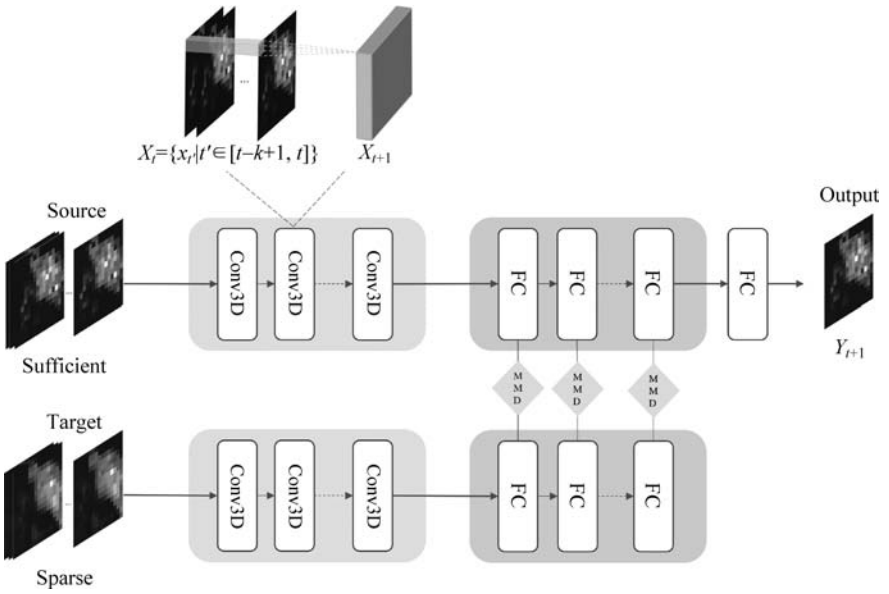

图 5-11　深度时空域适应网络模型框架

5.5.2　数据准备

本章使用 2015 年全年的 CitiBike 和 NYCTaxi 数据，以 1 小时为时间粒度，按照上述数据预处理生成区域级的出入流数据，使用过去 3 个时间步预测未来 1 个时间步的数据，对于源域，选择前 300 天的数据为训练集，对于目标域在前 300 天中随机选取 64 个时间片的数据进行训练，此后 32 天的数据作为验证集，最后 32 天的数据为测试集。

首先，整合 12 个月的数据，制作所需的数据集，并进行归一化，改写 torch. utils. data. Dataset 中的__getitem__()函数和__len__()函数来载入自己的数据集。并在创建完 Dataset 类之后，与 DataLoader 一起使用，以在模型训练时不断为模型提供数据。关于改写 Dataset 类，前述章节已有表述，本章不再赘述。

源域数据集 NYCTaxi 制作细节如下，目标域数据集制作只需更改数据存储目录 data_path，如 data_path='./CitiBike'，并将 np. fromfile()函数中 taxi 更改为 bike，代码如下。

```
import torch
import numpy as np

def make_dataset(data_duration = 3, label_duration = 1, data_path = './NYCTaxi'):
    flow_data_list = []
    x_data_list = []
    y_data_list = []
    for mon in range(1, 13):
        if mon < 10:
```

```
            date_str = str(2015) + '0' + str(mon)
        else:
            date_str = str(2015) + str(mon)
        flow_data_list.append(np.fromfile(data_path + '/taxi_%s' % date_str))

    raw_flow_data = np.concatenate(flow_data_list, axis=0).reshape((-1, 32, 32, 2))
    raw = (raw_flow_data - raw_flow_data.min()) / (raw_flow_data.max() - raw_flow_
data.min()) #归一化
    print('Flow data read over!!!')
    for i in range(data_duration, len(raw)):
        x_data = raw[i-data_duration:i]
        x_data = x_data.reshape(1, -1, raw.shape[1], raw.shape[2], 2)
        #print(x_data.shape)
        x_data_list.append(x_data)

        y_data = raw[i:i+label_duration]
        y_data = y_data.reshape(1, -1, raw.shape[1], raw.shape[2], 2)
        y_data_list.append(y_data)
    x_data_list = np.concatenate(x_data_list, axis=0)
    y_data_list = np.concatenate(y_data_list, axis=0)
    return x_data_list, y_data_list
```

改写源域的 Dataset 类，代码如下。

```
from torch.utils.data import Dataset

class NYCTaxiDataset(Dataset):
    def __init__(self, mode=None, split=None, data_path='./CitiBike'):
        people_data_x, people_data_y = make_dataset(data_path=data_path)
        people_data_x = people_data_x.transpose(0, 4, 1, 2, 3)
        people_data_y = people_data_y.transpose(0, 4, 1, 2, 3)
        self.mode = mode
        self.split = split

        if self.split is None:
            self.split = [300 * 24, 32 * 24, 32 * 24]

        if self.mode == 'train':
            self.x_data = people_data_x[0:self.split[0]]
            self.y_data = people_data_y[0:self.split[0]]
        elif self.mode == 'validate':
            self.x_data = people_data_x[self.split[0]:self.split[0] + self.split[1]]
            self.y_data = people_data_y[self.split[0]:self.split[0] + self.split[1]]
        elif self.mode == 'test':
            self.x_data = people_data_x[self.split[0] + self.split[1]:]
            self.y_data = people_data_y[self.split[0] + self.split[1]:]
```

```
    def __len__(self):
        return len(self.x_data)

    def __getitem__(self, item):
        self.sample_x = self.x_data[item]
        self.sample_y = self.y_data[item]
        return self.sample_x, self.sample_y
```

在本节中，假设目标域数据稀疏，所以随机在训练集中取 64 个时间片段的数据进行训练，随机选取，代码如下。

```
sample_list = []
count = 0
while count < 64:
    temInt = np.random.randint(0, 300 * 24)
    if temInt not in sample_list:
        sample_list.append(int(temInt))
        count += 1
sample_list.sort()
```

则改写目标域 Dataset 类的代码如下。

```
class CitiBikeDataset(Dataset):
    def __init__(self, mode = None, split = None):
        people_data_x, people_data_y = make_dataset(data_path = './DivvyBike')
        people_data_x = people_data_x.transpose(0, 4, 1, 2, 3)
        people_data_y = people_data_y.transpose(0, 4, 1, 2, 3)
        self.mode = mode
        self.split = split

        if self.split is None:
            self.split = [300 * 24, 32 * 24, 32 * 24]

        if self.mode == 'train':
            self.x_data = []
            self.y_data = []
            for i in sample_list:
                self.x_data.append(people_data_x[0:self.split[0]][i:i + 1])
                self.y_data.append(people_data_y[0:self.split[0]][i:i + 1])
            self.x_data = np.concatenate(self.x_data, 0)
            self.y_data = np.concatenate(self.y_data, 0)

        elif self.mode == 'validate':
            self.x_data = people_data_x[self.split[0]:self.split[0] + self.split[1]]
```

```
            self.y_data = people_data_y[self.split[0]:self.split[0] + self.split[1]]
        elif self.mode == 'test':
            self.x_data = people_data_x[self.split[0] + self.split[1]:]
            self.y_data = people_data_y[self.split[0] + self.split[1]:]

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

    def __getitem__(self, item):
        self.sample_x = self.x_data[item]
        self.sample_y = self.y_data[item]
        return self.sample_x, self.sample_y
```

构建完上述 Dataset 数据集，本章所需的源域和目标域 DataLoader 构建代码如下。

```
from torch.utils.data import DataLoader
src_train_dataset = CitiBikeDataset(mode = 'train')
src_train_dataloader = DataLoader(dataset = src_train_dataset, batch_size = 32, shuffle =
True)

tgt_train_dataset = DivvyBikeDataset(mode = 'train')
tgt_train_dataloader = DataLoader(dataset = tgt_train_dataset, batch_size = 32, shuffle =
False)

tgt_validate_dataset = DivvyBikeDataset(mode = 'validate')
tgt_validate_dataloader = DataLoader(dataset = tgt_validate_dataset, batch_size = 32,
shuffle = False)

tgt_test_dataset = DivvyBikeDataset(mode = 'test')
tgt_test_dataloader = DataLoader(dataset = tgt_test_dataset, batch_size = 32, shuffle =
False)
```

5.5.3　模型构建

模型构建方面，本节主要构建三个网络 SharedNet、MMDNet 与 PredictNet。

首先，建立 SharedNet，使用堆叠的 3D 卷积层学习原始出入流“图像”序列的数据表示，3D 卷积可以用于同时捕获空间与时间相关性。因此可以利用 3D 卷积层对出入流“图像”序列数据进行表示学习，以编码时空依赖，所有的 3D 卷积层由两个域的数据共享，从而将源域和目标域嵌入到一个共同的潜在表示空间中，本节在 SharedNet 中使用五层 3D 卷积层，可视实际情况加减层数，代码如下。

```
import torch
import torch.nn as nn


class SharedNet(nn.Module):
```

```
    def __init__(self):
        super(SharedNet, self).__init__()
        self.conv3d_1 = nn.Conv3d(in_channels = 2, out_channels = 16, kernel_size = 3,
stride = 1, padding = 1)
        self.relu_1 = nn.ReLU()

        self.conv3d_2 = nn.Conv3d(in_channels = 16, out_channels = 32, kernel_size = 3,
stride = 1, padding = 1)
        self.relu_2 = nn.ReLU()

        self.conv3d_3 = nn.Conv3d(in_channels = 32, out_channels = 64, kernel_size = 3,
stride = 1, padding = (0, 1, 1))
        self.relu_3 = nn.ReLU()

        self.conv3d_4 = nn.Conv3d(in_channels = 64, out_channels = 32, kernel_size = 3,
stride = 1, padding = 1)
        self.relu_4 = nn.ReLU()

        self.conv3d_5 = nn.Conv3d(in_channels = 32, out_channels = 2, kernel_size = (1, 3,
3), stride = 1, padding = (0, 1, 1))
        self.relu_5 = nn.ReLU()

    def forward(self, x):
        x = self.relu_1(self.conv3d_1(x))
        x = self.relu_2(self.conv3d_2(x))
        x = self.relu_3(self.conv3d_3(x))
        x = self.relu_4(self.conv3d_4(x))
        x = self.relu_5(self.conv3d_5(x))
        return x
```

为了迁移两个域之间的知识，笔者借助深度自适应网络(DAN)的思想，将由 3D 卷积层学到的源域与目标域的特征输入到本节建立的深度自适应网络 MMDNet 中，MMDNet 由堆叠的全连接网络组成，目的在于学习每个域的特征，同时，使用最大均值差异(MMD)作为约束进行知识迁移，为了学习两个域之间的可迁移特征，MMDNet 将每个域的数据表示嵌入到可再生希尔伯特空间中，然后将最大均值差异(MMD)作为约束，通过梯度下降与反向传播算法，将在可再生希尔伯特空间中源域与目标域的分布距离拉近，达到知识迁移的目的。本节使用一层全连接网络作为展示，读者可视情况加减，代码如下。

```
class MMDNet(nn.Module):
    def __init__(self):
        super(MMDNet, self).__init__()
        self.fc1 = nn.Linear(in_features = 32 * 32 * 2, out_features = 128)
        self.relu1 = nn.ReLU()

    def forward(self, x):
```

```
        x = x.view(x.shape[0], -1)
        x = self.relu1(self.fc1(x))
        return x
```

其中，最大均值差异（MMD）代码如下。

```
def DAN(source, target, kernel_mul=2.0, kernel_num=5, fix_sigma=None):
    batch_size = int(source.size()[0])
    kernels = guassian_kernel(source, target,
        kernel_mul=kernel_mul, kernel_num=kernel_num, fix_sigma=fix_sigma)

    loss1 = 0
    for s1 in range(batch_size):
        for s2 in range(s1+1, batch_size):
            t1, t2 = s1+batch_size, s2+batch_size
            loss1 += kernels[s1, s2] + kernels[t1, t2]
    loss1 = loss1 / float(batch_size * (batch_size - 1) / 2)

    loss2 = 0
    for s1 in range(batch_size):
        for s2 in range(batch_size):
            t1, t2 = s1+batch_size, s2+batch_size
            loss2 -= kernels[s1, t2] + kernels[s2, t1]
    loss2 = loss2 / float(batch_size * batch_size)
    return loss1 + loss2
```

最后，将源域特征输入 PredictNet 进行预测，PredictNet 由一层全连接网络组成，读者可视情况进行层数加减，代码如下。

```
class PredictNet(nn.Module):
    def __init__(self):
        super(PredictNet, self).__init__()
        self.fc1 = nn.Linear(in_features=128, out_features=32 * 32 * 2)
        self.relu1 = nn.ReLU()

    def forward(self, x):
        x = self.relu1(self.fc1(x))
        x = x.view(x.shape[0], 2, 1, 32, 32)
        return x
```

5.5.4　模型训练及测试

模型搭建完成后，本节将介绍如何训练神经网络，并对模型进行验证，以保存相对最优模型。首先，设定超参数，如学习率等，将 Epoch 总数设置为 200；对每一个 Epoch，在训练集上，按批次将数据输入网络学习特征，并进行预测，将预测的结果与真实值进行比较，利用设计好的损失函数与梯度下降方法优化模型；然后，每隔几个 Epoch 对模型进行

验证，如果结果比上次更好，则保存更好的模型。最后，用保存的模型对测试集进行测试，并记录结果。本节将均方根误差 RMSE 和最大均值差异加权求和作为损失函数，使用 Adam 优化器对模型进行优化。为了不与前述章节重复，本节仅使用 RMSE 作为模型的评价指标，但也可使用 MAE 与 WMAPE 作为评价指标，同时，为方便理解，本节也不设模型终止内容，读者可自行查阅前述章节，代码如下。

```
import numpy as np
import time
import torch
import torch.nn as nn
import torch.optim as optim

batch_size = 32

#使用 GPU 或 CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

len_src_train = len(src_train_dataloader)
len_tgt_train = len(tgt_train_dataloader)

lr = 0.00001
l2_decay = 5e-4
num_epoches = 200

src_loss_list = []
total_loss_list = []
tgt_val_loss_list = []

seed = 32
np.random.seed(seed=seed)
torch.manual_seed(seed)
if device == 'cuda':
    torch.cuda.manual_seed(seed)

task_criterion = nn.MSELoss()

BaseNet = SharedNet().to(device)
TransferNet = MMDNet().to(device)
TaskNet = PredictNet().to(device)

for param in BaseNet.parameters():
    param.requires_grad = False

optimizer = optim.Adam([
    {'params': BaseNet.parameters()},
    {'params': TransferNet.parameters()},
    {'params': TaskNet.parameters()}], lr=lr, weight_decay=l2_decay)
best_rmse = 100000

for epoch in range(num_epoches):
```

```
    t0 = time.time()
    BaseNet.train()
    TransferNet.train()
    TaskNet.train()

    src_train_aver_rmse = 0
    mmd_loss = 0

    iter_src = iter(src_train_dataloader)
    iter_tgt = iter(tgt_train_dataloader)

    num_iter = len_src_train
    #模型训练
    for i in range(0, num_iter):
        src_data_x, src_data_y = next(iter_src)
        tgt_data_x, tgt_data_y = next(iter_tgt)

        if (i + 1) % len_tgt_train == 0:
            iter_tgt = iter(tgt_train_dataloader)

        src_data_x = src_data_x.float().to(device)
        src_data_y = src_data_y.float().to(device)
        tgt_data_x = tgt_data_x.float().to(device)
        tgt_data_y = tgt_data_y.float().to(device)

        optimizer.zero_grad()

        inputs = torch.cat((src_data_x, tgt_data_x), dim = 0)
        features = BaseNet(inputs)
        features = TransferNet(features)
        outputs = TaskNet(features)

        #print(outputs.shape, src_data_y.shape, inputs.size(0)/2)

        task_loss = torch.sqrt(task_criterion(outputs.narrow(0, 0, int(inputs.size(0) /
2)), src_data_y))

        transfer_loss = DAN(features.narrow(0, 0, int(features.size(0) / 2)),
                            features.narrow(0, int(features.size(0) / 2), int(features.
size(0) / 2)))

        total_loss = 0.1 * transfer_loss + task_loss

        src_train_aver_rmse += total_loss.item()
        mmd_loss += transfer_loss.item()
        total_loss.backward()
        optimizer.step()
    src_train_aver_rmse /= len_src_train
```

```
        mmd_loss /= len_src_train
        #模型验证
        if (epoch + 1) % 5 == 0 or epoch == 0:
            BaseNet.eval()
            TransferNet.eval()
            TaskNet.eval()
            tgt_validate_aver_rmse = 0
            len_tgt_validate = len(tgt_validate_dataloader)
            for i, (tgt_data_x, tgt_data_y) in enumerate(tgt_validate_dataloader):
                tgt_data_x, tgt_data_y = tgt_data_x.float().to(device), tgt_data_y.float
    ().to(device)
                features = TransferNet(BaseNet(tgt_data_x))
                tgt_output = TaskNet(features)
                tgt_loss = torch.sqrt(task_criterion(tgt_output, tgt_data_y))
                tgt_validate_aver_rmse += tgt_loss.item()
            tgt_validate_aver_rmse /= len_tgt_validate
            if tgt_validate_aver_rmse < best_rmse:
                best_rmse = tgt_validate_aver_rmse
                torch.save(BaseNet.state_dict(), 'best_BaseNet_transfer_model.pkl')
                torch.save(TransferNet.state_dict(), 'best_TransferNet_model.pkl')
                torch.save(TaskNet.state_dict(), 'best_TaskNet_model.pkl')
        t1 = time.time()
        print('Epoch: [{}/{}], Source train loss: {}, MMD loss: {}, tgt_best_validate_loss: {},
    Cost {}min.'.format(epoch + 1, num_epoches, src_train_aver_rmse, mmd_loss, best_rmse, (t1
    - t0) / 60))
```

模型测试部分代码与上述代码中模型验证部分类似，但稍有不同。测试过程首先需要利用 torch.load()函数将保存于本地的最佳模型重新导入，然后利用 load_state_dict()函数将保存的参数字典加载到模型中，此时，模型中的参数即为训练过程中训练好的参数。代码过程可以参考前述章节模型加载部分与上述模型验证部分代码，本节不再赘述。

5.5.5 结果展示

SharedNet 模型示例如图 5-12 所示。

```
SharedNet(
  (conv3d_1): Conv3d(2, 16, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
  (relu_1): ReLU()
  (conv3d_2): Conv3d(16, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
  (relu_2): ReLU()
  (conv3d_3): Conv3d(32, 64, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(0, 1, 1))
  (relu_3): ReLU()
  (conv3d_4): Conv3d(64, 32, kernel_size=(3, 3, 3), stride=(1, 1, 1), padding=(1, 1, 1))
  (relu_4): ReLU()
  (conv3d_5): Conv3d(32, 2, kernel_size=(1, 3, 3), stride=(1, 1, 1), padding=(0, 1, 1))
  (relu_5): ReLU()
)
```

图 5-12 SharedNet 模型示例

MMDNet 模型示例如图 5-13 所示。

```
MMDNet(
  (fc1): Linear(in_features=2048, out_features=128, bias=True)
  (relu1): ReLU()
)
```

图 5-13　MMDNet 模型示例

PredictNet 模型示例如图 5-14 所示。

```
PredictNet(
  (fc1): Linear(in_features=128, out_features=2048, bias=True)
  (relu1): ReLU()
)
```

图 5-14　PredictNet 模型示例

模型训练过程展示示例如图 5-15 所示。

```
Epoch: [1/200], Source train loss: 0.19903759216820752, MMD loss: 1.5250267894179732,
tgt_best_validate_loss: 0.036864012479782104, Cost 0.2752910017967224min.
Epoch: [2/200], Source train loss: 0.14947643200004543, MMD loss: 1.0328753568508007,
tgt_best_validate_loss: 0.036864012479782104, Cost 0.26984686851501466min.
Epoch: [3/200], Source train loss: 0.11820515093428118, MMD loss: 0.7233671170693857,
tgt_best_validate_loss: 0.036864012479782104, Cost 0.2812255581219991min.
Epoch: [4/200], Source train loss: 0.09960045900057864, MMD loss: 0.5405430175639965,
tgt_best_validate_loss: 0.036864012479782104, Cost 0.28381844361623126min.
Epoch: [5/200], Source train loss: 0.08962199381656116, MMD loss: 0.4438382696222376,
tgt_best_validate_loss: 0.035327720393737154, Cost 0.2979815642038981min.
Epoch: [6/200], Source train loss: 0.08198306323201568, MMD loss: 0.3705919142122622,
tgt_best_validate_loss: 0.035327720393737154, Cost 0.3461944778760274min.
Epoch: [7/200], Source train loss: 0.07650970833169089, MMD loss: 0.31895606606094923,
tgt_best_validate_loss: 0.035327720393737154, Cost 0.3792957623799642min.
Epoch: [8/200], Source train loss: 0.07162327740203452, MMD loss: 0.2731713365625452,
tgt_best_validate_loss: 0.035327720393737154, Cost 0.3475567102432251min.
```

图 5-15　模型训练过程展示示例

本案例仅进行简要的展示，供读者参考，模型并未进行细致的调参，也并未进行模型结果的调试，仅凭笔者经验构建本模型，读者可在此模型基础上进行细致的调参，并借鉴相关资料进行模型结构调整，以提高模型性能表现。

完整的代码可通过本书提供的获取方式进行获取。

5.6　本章小结

本章首先介绍了人工智能在共享单车大数据上应用的重要性与必要性，介绍了几个主流研究方向，并以基于深度域适应网络的共享单车出入流迁移学习为研究案例，详细介绍了共享单车数据的获取手段及开源数据集、数据预处理过程以及深度模型的建立、训练和测试过程。通过对本章的学习，初学者可以对共享单车出入流预测有一个直观的了解。