# 3.4-A 模式生成：从聚类模式特征到生成对抗网络和计算分析工具的建构

## 3.4.1 聚类方式模式提取

寻找地理空间数据分布的特征、模式组成和结构，在*标记距离*一章通过构建类/簇大小直方图、共现关系和层级分解等标记特征提取土地覆盖类型的分布模式。聚类则可以把相似的样本（或空间栅格单元、点、区域等）聚为同类（簇），同簇样本具有相似的属性特征，不同簇的样本之间则为高度不同的属性特征，因此分析聚类后簇内样本属性特征的组成和结构可以寻找地理空间数据分布的模式。

地理空间数据聚类的空间方式一般有两种，一种是直接将具有多种属性的栅格单元作为样本用于聚类；另一种是按照样方区域统计多元栅格数据后执行聚类。如果仅分析土地覆盖类型一种分类数据源属性，则采用先区域统计（例如土地利用类型频数）再聚类的方式，发现采样样方具有的类型组成模式。

关于聚类的详细解释可以参考*机器学习实验*一章。

### 3.4.1.1 基于样方的多源栅格数据区域统计

多源数据通常具有不同的数据格式，不同的高空分辨率，不同的范围，因此需要将其规范为具有同样单元大小和同一分析范围的栅格数据。在获得原始数据之后，需要先查看数据，可以使用 QGIS 工具，下述也定义了tiled\_web\_map\_show()方法，以金子塔缩放（zoom）瓦片网络地图形式（tiled web map）显示数据，提高大数据显示效率。

下述实验了6类数据，如表：

| 序号 | 数据名称 | 数据来源 | 数据说明 | 大小 | 数据类型 | 时间 | 精度 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 建筑高度（层高）数据 （Building Footprints (current) ） | [Chicago Data Portal](https://data.cityofchicago.org/Buildings/Building-Footprints-current-/hz9b-7nh8)① | 为芝加哥城建筑轮廓数据，含有层数等字段 | 5.96GB | JSON | 2021.08.21 | 矢量 |
| 2 | 土地利用数据（Land Use Inventory） | [Chicago Metropolitan Agency for Planning, CMAP](https://www.cmap.illinois.gov/data/land-use/inventory)② | 分类信息查看*不平等性和空间隔离*一章。原始数据为SHP格式 | 2.09GB | TIFF（SHP） | 2015 | 5m（矢量） |
| 3 | 土地覆盖数据（High-resolution, 7-class, 2010 land cover for the Chicago, IL region.） | [ArcGIS-Chicago Regional Land Cover](https://www.arcgis.com/home/item.html?id=782adcff882d4f09a227b509dcaa1628)③ | 土地覆盖类型说明：0 - Background；1 - Tree Canopy；2 - Grass/Shrub；3 - Bare Soil；4 - Water；5 - Buildings；6 - Roads/Railroads；7 - Other Paved Surfaces | 17.8GB | TIFF | 2010 | 1m |
| 4 | 地表温度数据（Land surface temperature，LST）(MODIS MYD21A1D v006) | [USGS EarthExploer](https://earthexplorer.usgs.gov/)④ | 选取了冬季和夏季区间的两个数据 | 395MB403MB | TIFF | 2022.02.262022.08.10 | 1km |
| 5 | DEM（Digital Elevation Model） 数据（GTOPO30） | [USGS EarthExploer](https://earthexplorer.usgs.gov/)④ | 含两个数据（用于合并）：n41\_w088\_1arc\_v3.tifn42\_w088\_1arc\_v3.tif | 24.7MB24.7MB | TIFF | 1996 | 1km |
| 6 | 夜间灯光数据（Annual VNL V2） | [Earth Observation Group](https://eogdata.mines.edu/products/vnl/)⑤ | VNL\_v21\_npp\_2021\_global\_vcmslcfg\_c202205302300.average.dat.tif，为2021年的均值数据 | 10.8GB | TIFF | 2021 | ~500m |

# IPython extension to reload modules before executing user code.  
%load\_ext autoreload   
# Reload all modules (except those excluded by %aimport) every time before executing the Python code typed.  
%autoreload 2   
  
import warnings  
warnings.filterwarnings('ignore')  
  
from usda.utils import AttrDict  
import usda.database as usda\_database  
import usda.geodata\_process as usda\_geoprocess  
import usda.models as usda\_model   
import usda.pattern\_signature as usda\_signature  
from usda import data\_visualization as usda\_vis  
from usda import network as usda\_network  
from scipy.stats import zscore  
  
import matplotlib.pyplot as plt  
import matplotlib  
import rasterio as rio  
import earthpy.plot as ep  
import copy  
import numpy as np  
from rio\_tiler.io import COGReader  
import geopandas as gpd  
import pandas as pd  
import seaborn as sns

The autoreload extension is already loaded. To reload it, use:  
 %reload\_ext autoreload

\_\_C=AttrDict()   
args=\_\_C  
  
\_\_C.gi=AttrDict()  
\_\_C.gi.epsg\_wgs84=4326  
\_\_C.gi.Chicago\_epsg=32616  
  
\_\_C.data=AttrDict()  
\_\_C.data.footprint\_wgs84\_fn='G:\data\Building\_Footprints\Building\_Footprints.shp'  
\_\_C.data.footprint\_projected\_fn='G:\data\Building\_Footprints\Building\_Footprints\_projected.shp'  
\_\_C.data.stories\_fn='G:\data\Building\_Footprints\chicago\_building\_stories.tif'  
  
\_\_C.data.landuse\_fn='E:\data\Chicago\_landuse\landuse.tif'   
\_\_C.data.landcover\_fn='F:\data\data\_01\_Chicago\LandCover\_2010\_ChicagoRegion\landcover\_2010\_chicagoregion.img'  
\_\_C.data.lst\_0226\_fn=r'E:\data\LST\lst\_0226\_chicago.tif' # <MDI key="RANGEBEGINNINGDATE">2023-02-26</MDI> <MDI key="RANGEBEGINNINGTIME">00:00:00</MDI> <MDI key="RANGEENDINGDATE">2023-02-26</MDI> <MDI key="RANGEENDINGTIME">23:59:59</MDI>  
\_\_C.data.lst\_0810\_fn=r'E:\data\LST\lst\_0810\_chicago.tif' # <RangeEndingTime>23:59:59.000000</RangeEndingTime> <RangeEndingDate>2022-08-10</RangeEndingDate> <RangeBeginningTime>00:00:00.000000</RangeBeginningTime> <RangeBeginningDate>2022-08-10</RangeBeginningDate>  
\_\_C.data.nightlight\_fn=r'D:\data\night\_light\vnl\_chicago\_year.tif'  
  
\_\_C.data.mosaic\_dem\_root=r'G:\data\chicago\_dem\mosaic\_data'  
\_\_C.data.dem\_fn=r'G:/data/chicago\_dem/chicago\_dem.tif'  
  
\_\_C.data.chicago\_boundary\_fn=r'G:\data\Chicago\_boundaries\_city\Chicago\_boundaries\_city.shp'  
\_\_C.data.zs\_gdf\_fn=r'D:\data\zonal\_stats\_gdf.gpkg'  
\_\_C.data.zs\_norm\_gdf\_fn=r'D:\data\zonal\_stats\_norm\_gdf.gpkg'  
  
\_\_C.data.naip\_root=r'F:\data\NAIP\_chicago'

#### 1) 原始数据预处理与打印观察

* 建筑高度（层高）数据

定义json2gdf()方法将JSON格式数据读取为GeoDataFrame格式，并存储为SHP格式用于调用定义的create\_multiband\_raste()方法，将SHP转为栅格数据，栅格单元大小同一配置为10m，投影同一配置为EPSG:32616。

building\_footprints=usda\_database.json2gdf(args.data.footprint\_wgs84\_fn,numeric\_columns={'no\_stories':'int','stories':'int'},epsg=args.gi.Chicago\_epsg)  
building\_footprints.to\_file(args.data.footprint\_projected\_fn)

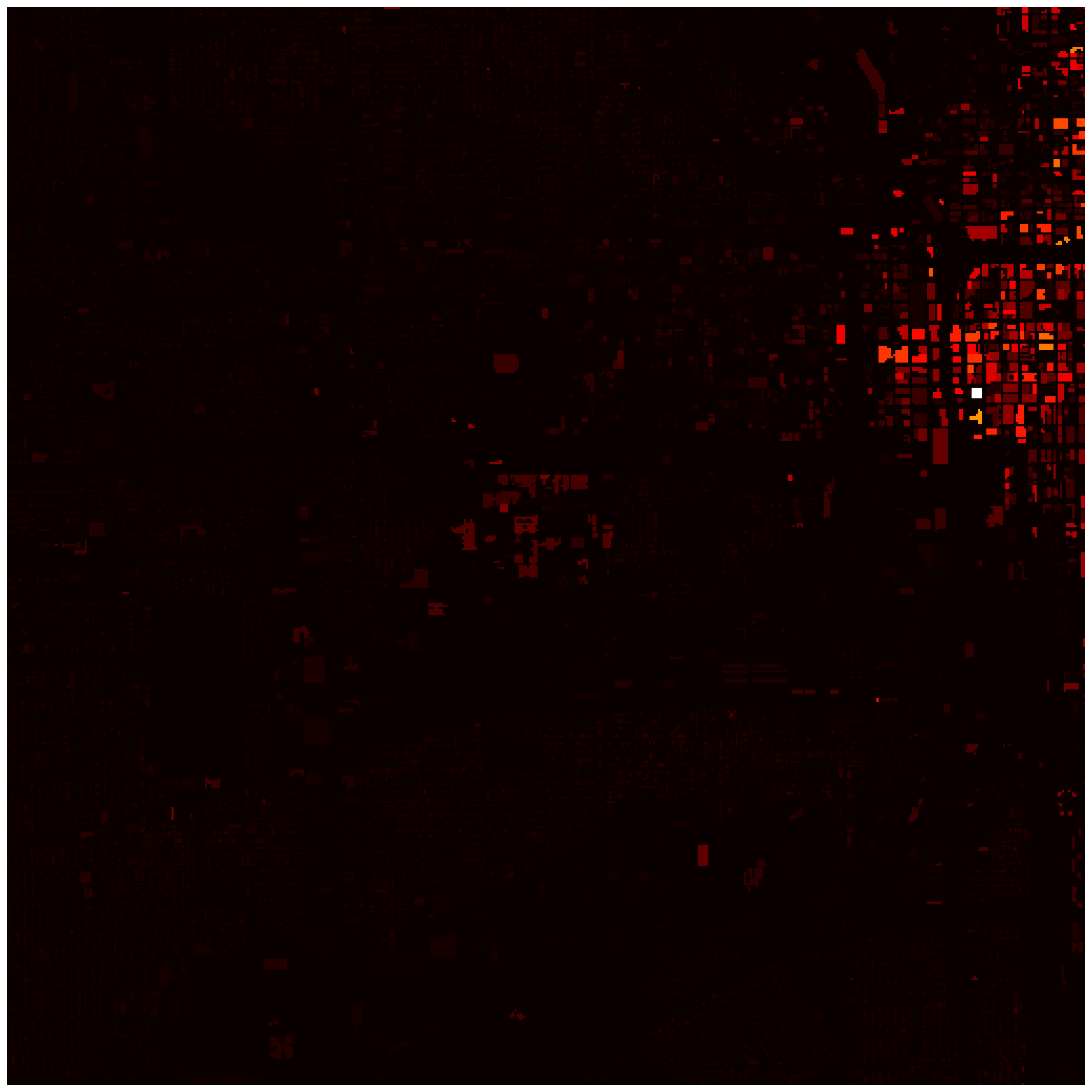
usda\_geoprocess.create\_multiband\_raster(['stories'],args.data.footprint\_projected\_fn,args.data.stories\_fn,cellSize=10,NoData\_value=0)

The raster was written successfully!

调用tiled\_web\_map\_show()方法打印地图是，可以配置球面墨卡托投影金字塔缩放比例z参数，为0-23级，显示的范围越大，数值越大。参数centroid\_latlon可以配置缩放的中心点，为经纬度。

centroid\_latlon=(41.887857,-87.668474)  
\_=usda\_geoprocess.tiled\_web\_map\_show(args.data.stories\_fn,z=12,cmap='hot',centroid\_latlon=centroid\_latlon,figsize=(20,20))

CRS: EPSG:32616  
影像边界坐标： (-87.94015009829613, 41.642095322269824, -87.52374193269772, 42.02378751932716)  
瓦片的形状： (512, 512, 1)

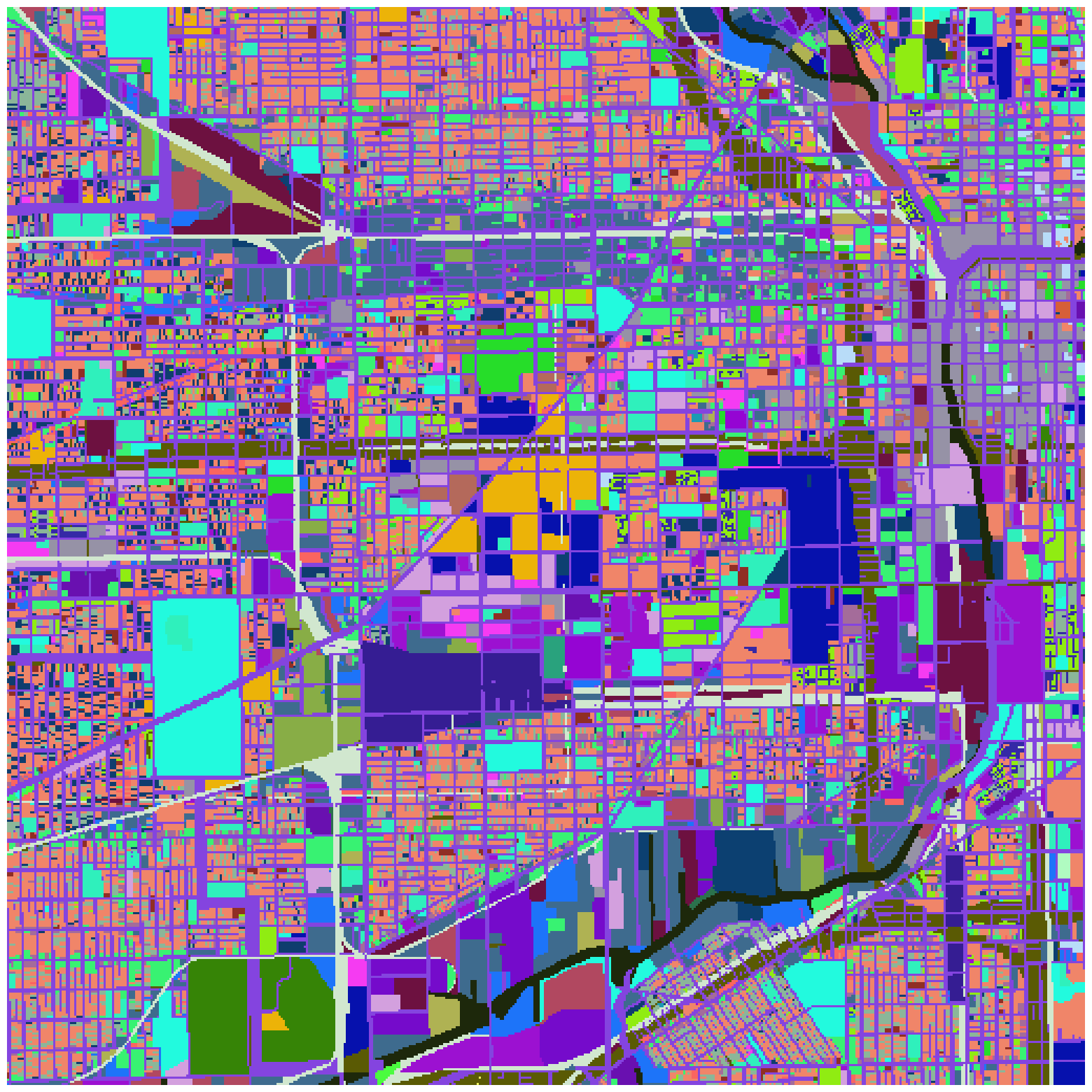


* 土地利用数据

使用*不平等性和空间隔离*一章处理后，单元大小为5m的栅格数据。

np.random.seed(0)  
cmap=matplotlib.colors.ListedColormap (np.random.rand(256,3))  
  
\_=usda\_geoprocess.tiled\_web\_map\_show(args.data.landuse\_fn,z=12,cmap=cmap,centroid\_latlon=centroid\_latlon,figsize=(20,20))

CRS: EPSG:32616  
影像边界坐标： (-88.71487753316414, 41.19605815460622, -87.51987719192424, 42.50552725246455)  
瓦片的形状： (512, 512, 1)

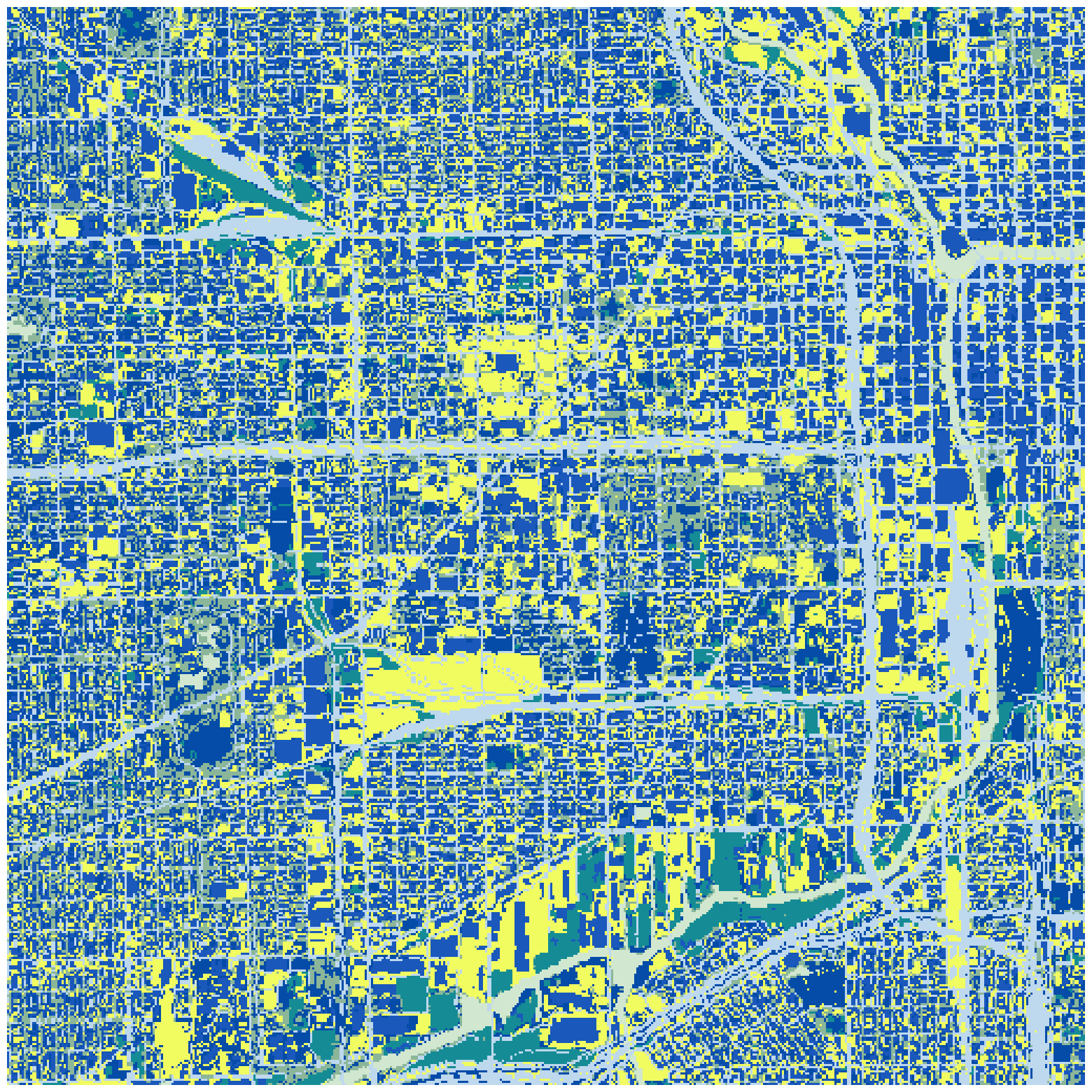


* 土地覆盖数据

原始数据为栅格数据，因此直接读取查看数据是否正确。

\_=usda\_geoprocess.tiled\_web\_map\_show(args.data.landcover\_fn,z=12,cmap=cmap,centroid\_latlon=centroid\_latlon,figsize=(20,20))

CRS: PROJCS["Albers Conical Equal Area",GEOGCS["NAD83",DATUM["North\_American\_Datum\_1983",SPHEROID["GRS 1980",6378137,298.257222101,AUTHORITY["EPSG","7019"]],AUTHORITY["EPSG","6269"]],PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9122"]],AUTHORITY["EPSG","4269"]],PROJECTION["Albers\_Conic\_Equal\_Area"],PARAMETER["latitude\_of\_center",23],PARAMETER["longitude\_of\_center",-96],PARAMETER["standard\_parallel\_1",29.0833333333333],PARAMETER["standard\_parallel\_2",45.0833333333333],PARAMETER["false\_easting",0],PARAMETER["false\_northing",0],UNIT["meters",1],AXIS["Easting",EAST],AXIS["Northing",NORTH]]  
影像边界坐标： (-88.84156993936557, 41.05502222871079, -86.28757940502697, 42.56699636977984)  
瓦片的形状： (512, 512, 1)

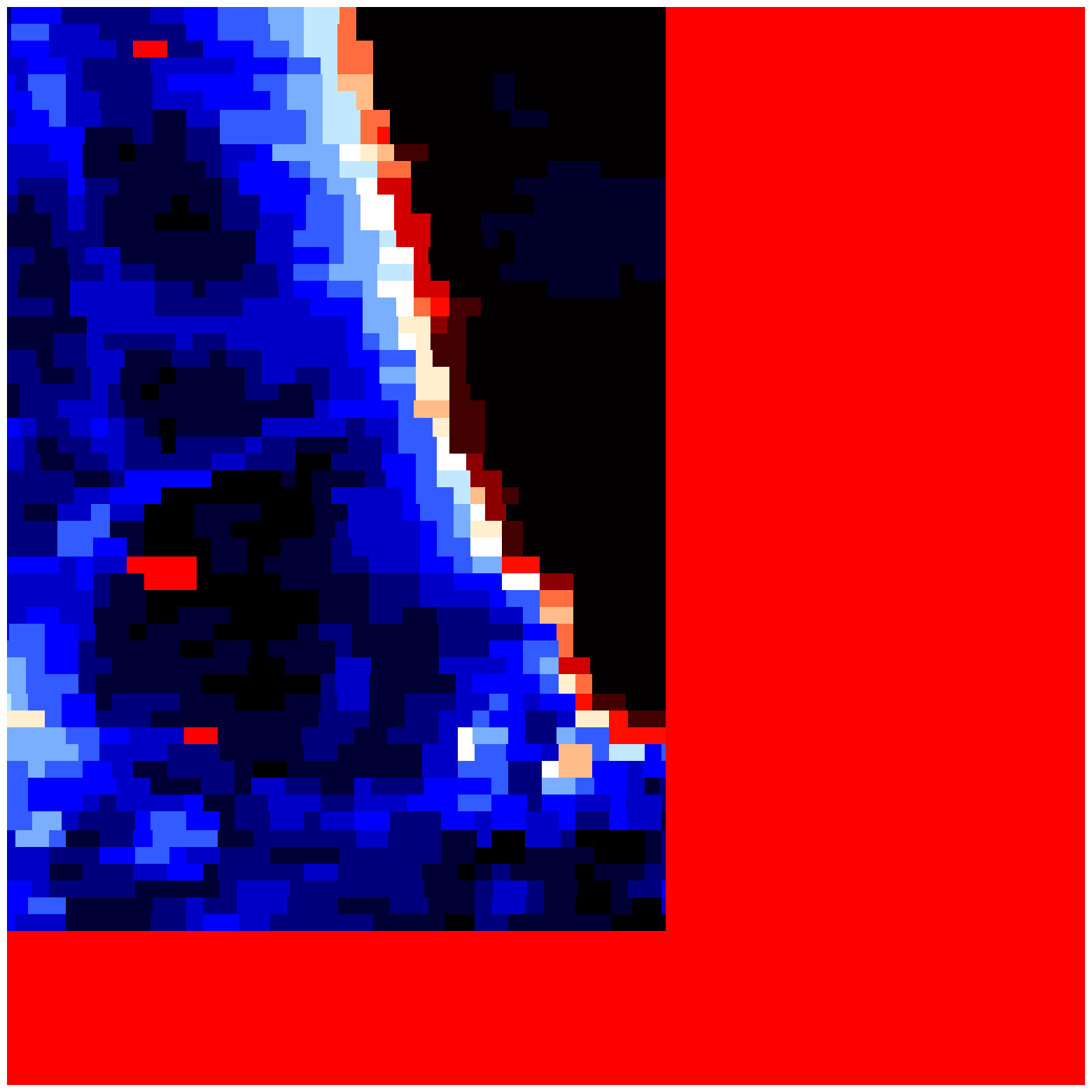


* 地表温度（LST） 数据

选用了夏季和冬季区域两个数据，用于观察夏季降温和冬季保温效果，数据分别标识为0810（夏季）和0226（冬季）。

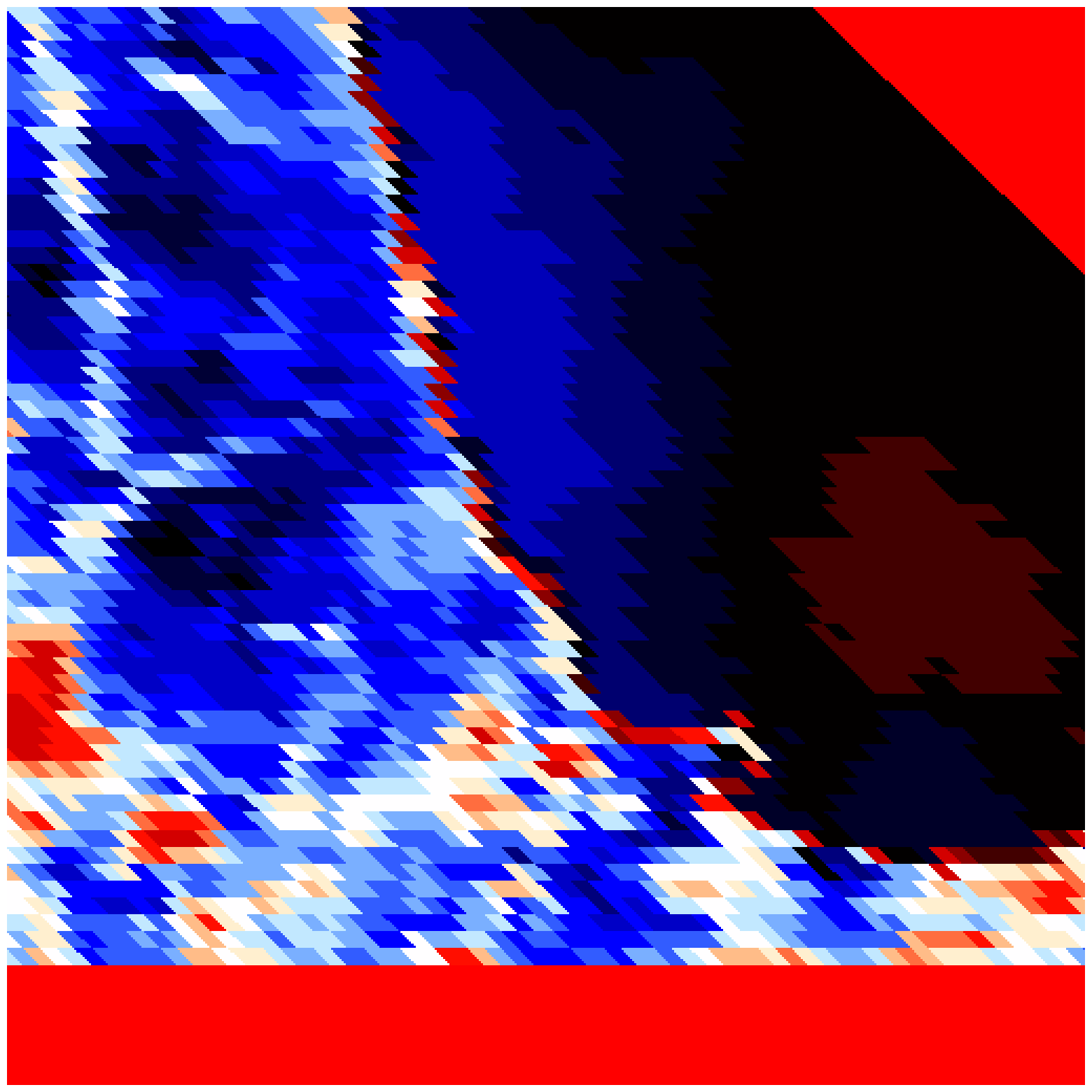
centroid\_latlon=(41.887857,-87.668474)  
lst\_tile=usda\_geoprocess.tiled\_web\_map\_show(args.data.lst\_0226\_fn,z=9,cmap='flag',centroid\_latlon=centroid\_latlon,figsize=(20,20))

CRS: EPSG:4326  
影像边界坐标： (-88.00282277592254, 41.583333329333335, -87.46106746458194, 42.09166666266667)  
瓦片的形状： (512, 512, 1)



lst\_tile=usda\_geoprocess.tiled\_web\_map\_show(args.data.lst\_0810\_fn,z=9,cmap='flag',centroid\_latlon=centroid\_latlon,figsize=(20,20))

CRS: PROJCS["unnamed",GEOGCS["Unknown datum based upon the custom spheroid",DATUM["Not specified (based on custom spheroid)",SPHEROID["Custom spheroid",6371007.181,0]],PRIMEM["Greenwich",0],UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9122"]]],PROJECTION["Sinusoidal"],PARAMETER["longitude\_of\_center",0],PARAMETER["false\_easting",0],PARAMETER["false\_northing",0],UNIT["metre",1,AUTHORITY["EPSG","9001"]],AXIS["Easting",EAST],AXIS["Northing",NORTH]]  
影像边界坐标： (-88.74908925022875, 41.5666666631397, -86.73220428347186, 42.091666663061076)  
瓦片的形状： (512, 512, 1)



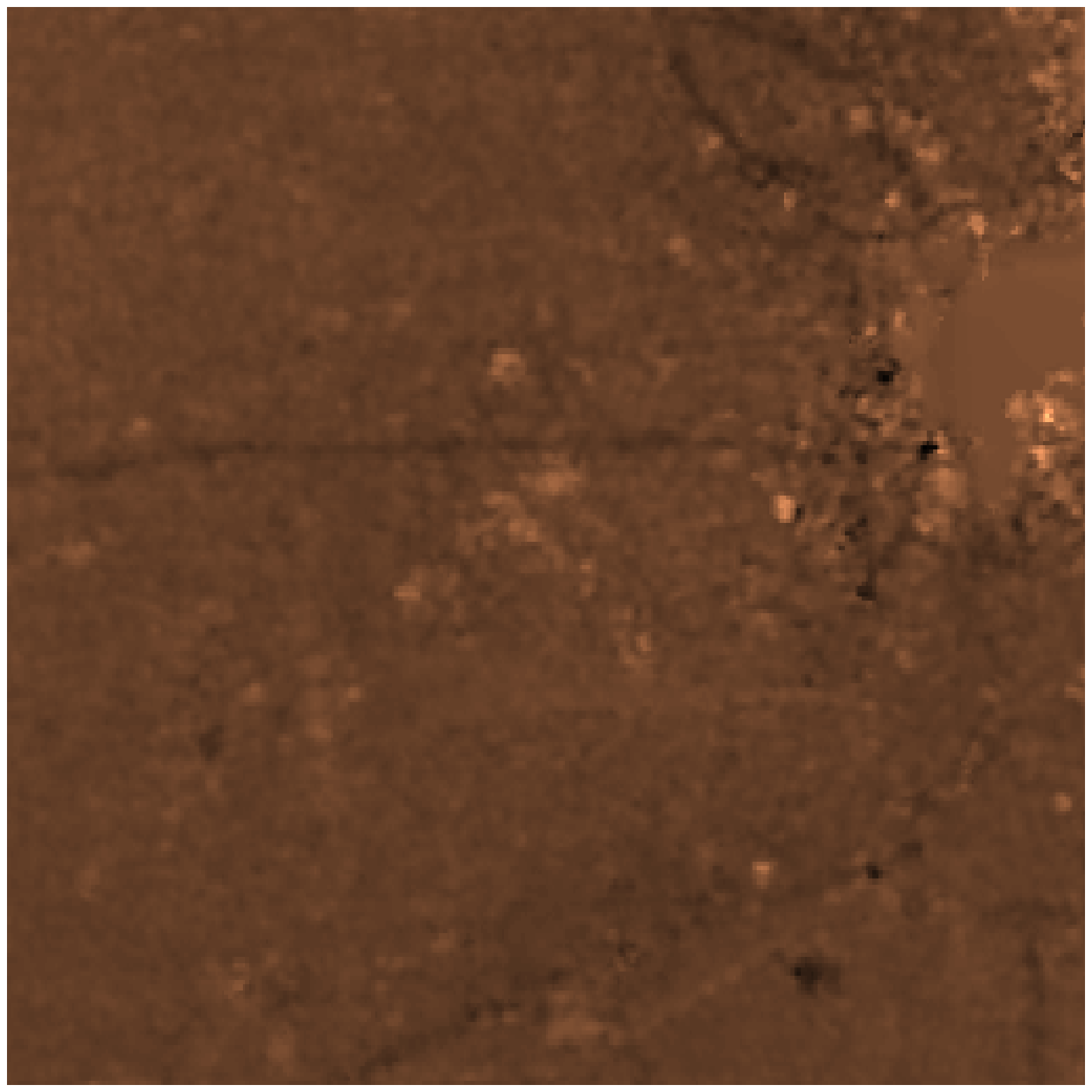
* DEM 数据

定义raster\_mosaic()方法合并两个DEM数据覆盖芝加哥城全部区域。

\_=usda\_geoprocess.raster\_mosaic(args.data.mosaic\_dem\_root,args.data.dem\_fn,dtype=rio.int32)

lst\_tile=usda\_geoprocess.tiled\_web\_map\_show(args.data.dem\_fn,z=12,cmap='copper',centroid\_latlon=centroid\_latlon,figsize=(20,20))

CRS: EPSG:4326  
影像边界坐标： (-88.00013888888888, 40.99986111111111, -86.9998611111111, 43.000138888888884)  
瓦片的形状： (512, 512, 1)



* 夜间灯光数据

nighlight\_tile=usda\_geoprocess.tiled\_web\_map\_show(args.data.nightlight\_fn,z=8,cmap='hot',centroid\_latlon=centroid\_latlon,figsize=(20,20))

CRS: EPSG:4326  
影像边界坐标： (-91.61041595954998, 39.147916379849995, -79.79791586504999, 44.97291642645)  
瓦片的形状： (512, 512, 1)



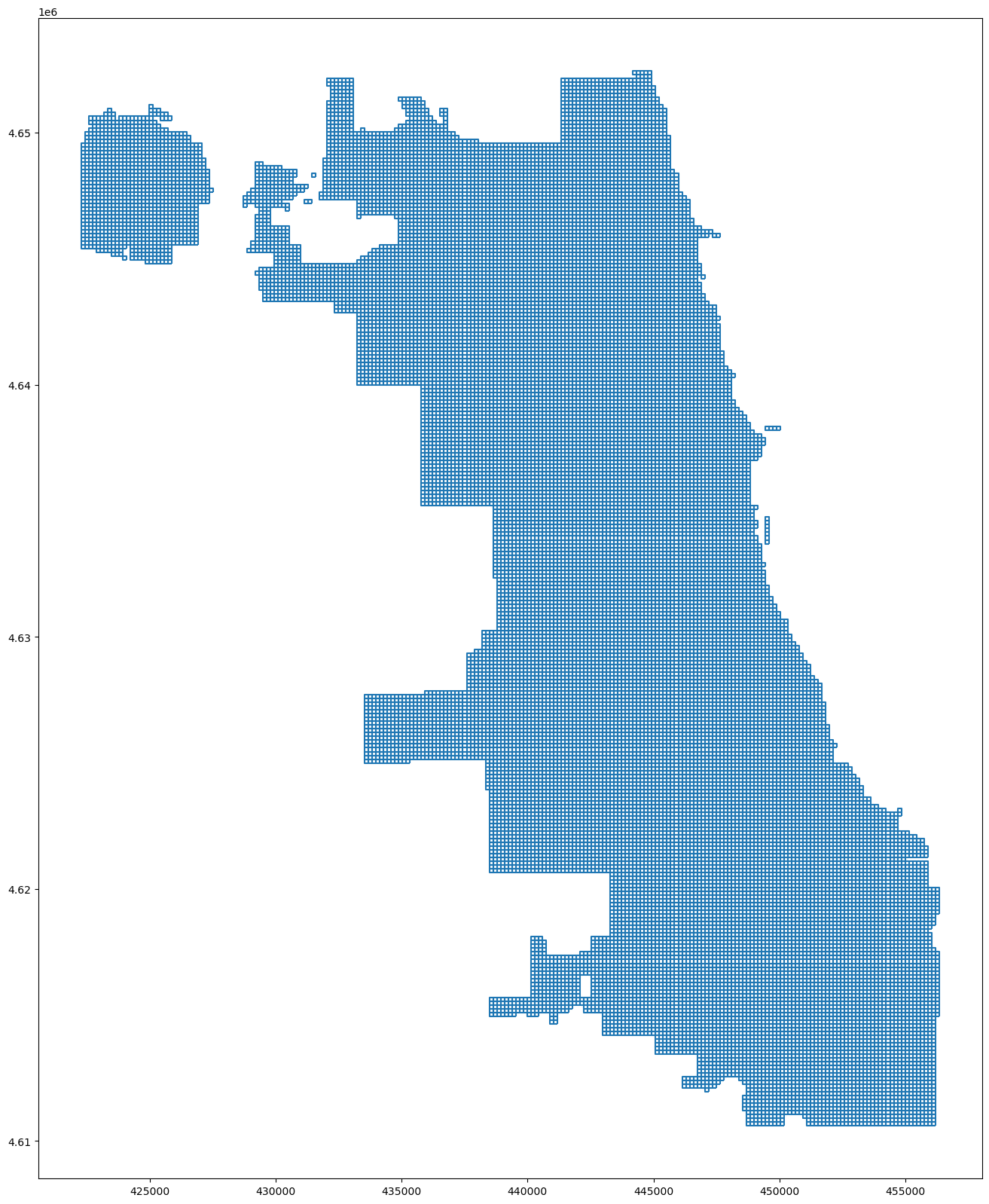
#### 2) 多源数据的区域统计与数据合并

定义rec\_quadrats\_bounded\_gdf()方法可以实现根据指定的区域构建样方，此次实验定义的样方大小为150m，基本为芝加哥城一个街区的大小。样方数为 25731 个。

boundary=gpd.read\_file(args.data.chicago\_boundary\_fn)  
boundary.to\_crs(args.gi.Chicago\_epsg,inplace=True)  
  
grids=usda\_geoprocess.rec\_quadrats\_bounded\_gdf(boundary,h\_distance=150,v\_distance=150)  
print(grids.shape)

(25731, 1)

grids.boundary.plot(figsize=(20,20));



多源数据的区域统计使用定义的zonal\_stats\_raster\_batch()方法实现，其参数raster\_info为一个字典，值列表中的第1个值为数据文件路径，第2个值为区域统计方法，可以传入一个包含多个统计方法的列表。区域统计的方法包括’count’, ‘min’, ‘max’, ‘mean’, ‘sum’, ‘std’, ‘median’, ‘majority’, ‘minority’, ‘unique’, ‘range’, ‘nodata’, ’nan’等，及自定义的频数统计’frequency’方式。对于分类数据（土地利用和覆盖类型数据）采用频数统计的方法，即一个样方中各类型的数量或占比；其它的数据按样方计算均值。基于样方的区域统计计算需要一段时间，将计算结果存储为GPKG格式于本地磁盘，避免重复计算。

raster\_info={   
 'lst0226':[args.data.lst\_0226\_fn,'mean'],  
 'lst0810':[args.data.lst\_0810\_fn,'mean'],  
 'stories':[args.data.stories\_fn,'mean'],   
 'dem':[args.data.dem\_fn,'mean'],  
 'nightlight':[args.data.nightlight\_fn,['mean']],  
 'landuse':[args.data.landuse\_fn,'frequency'],  
 'landcover':[args.data.landcover\_fn,'frequency'],  
 }  
  
zs\_gdf=usda\_geoprocess.zonal\_stats\_raster\_batch(raster\_info,grids,upscale\_mu=2)   
zs\_gdf.to\_file(args.data.zs\_gdf\_fn,driver='GPKG')

Processing img: 1/7-lst0226  
resampling upscale=9  
Processing img: 2/7-lst0810  
resampling upscale=9  
Processing img: 3/7-stories  
Processing img: 4/7-dem  
Processing img: 5/7-nightlight  
resampling upscale=6  
Processing img: 6/7-landuse  
Processing img: 7/7-landcover

区域统计多源数据于一个GeoDataFrame格式数据下，通过字段名标识不同数据统计结果。

zs\_gdf=gpd.read\_file(args.data.zs\_gdf\_fn)  
# zs\_gdf.fillna(0,inplace=True)  
print(zs\_gdf.shape)  
print(zs\_gdf.columns)  
zs\_gdf.head(3)

(25731, 71)  
Index(['on', 'lst0226\_mean', 'lst0810\_mean', 'stories\_mean', 'dem\_mean',  
 'nightlight\_mean', 'landuse\_fre\_31', 'landuse\_fre\_28', 'landuse\_fre\_55',  
 'landuse\_fre\_29', 'landuse\_fre\_10', 'landuse\_fre\_43', 'landuse\_fre\_54',  
 'landuse\_fre\_19', 'landuse\_fre\_42', 'landuse\_fre\_0', 'landuse\_fre\_2',  
 'landuse\_fre\_18', 'landuse\_fre\_4', 'landuse\_fre\_1', 'landuse\_fre\_12',  
 'landuse\_fre\_46', 'landuse\_fre\_14', 'landuse\_fre\_41', 'landuse\_fre\_8',  
 'landuse\_fre\_47', 'landuse\_fre\_30', 'landuse\_fre\_16', 'landuse\_fre\_9',  
 'landuse\_fre\_50', 'landuse\_fre\_40', 'landuse\_fre\_56', 'landuse\_fre\_23',  
 'landuse\_fre\_20', 'landuse\_fre\_33', 'landuse\_fre\_13', 'landuse\_fre\_32',  
 'landuse\_fre\_45', 'landuse\_fre\_6', 'landuse\_fre\_37', 'landuse\_fre\_27',  
 'landuse\_fre\_34', 'landuse\_fre\_26', 'landuse\_fre\_49', 'landuse\_fre\_24',  
 'landuse\_fre\_11', 'landuse\_fre\_15', 'landuse\_fre\_25', 'landuse\_fre\_48',  
 'landuse\_fre\_51', 'landuse\_fre\_7', 'landuse\_fre\_39', 'landuse\_fre\_5',  
 'landuse\_fre\_52', 'landuse\_fre\_53', 'landuse\_fre\_17', 'landuse\_fre\_38',  
 'landuse\_fre\_22', 'landuse\_fre\_35', 'landuse\_fre\_36', 'landuse\_fre\_44',  
 'landuse\_fre\_3', 'landcover\_fre\_2', 'landcover\_fre\_7',  
 'landcover\_fre\_5', 'landcover\_fre\_6', 'landcover\_fre\_1',  
 'landcover\_fre\_4', 'landcover\_fre\_3', 'landcover\_fre\_0', 'geometry'],  
 dtype='object')

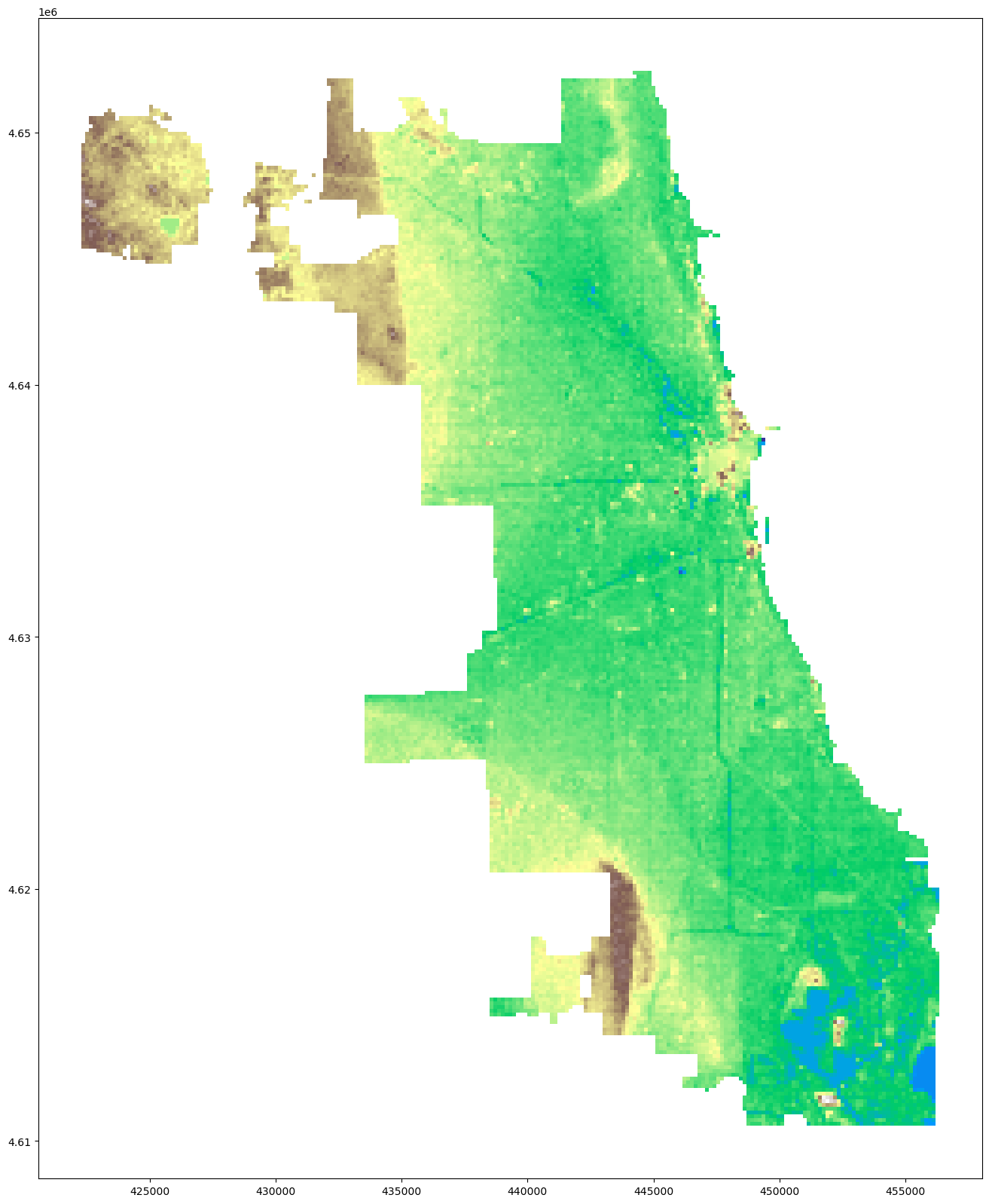
|  | **on** | **lst0226\_mean** | **lst0810\_mean** | **stories\_mean** | **dem\_mean** | **nightlight\_mean** | **landuse\_fre\_31** | **landuse\_fre\_28** | **landuse\_fre\_55** | **landuse\_fre\_29** | **...** | **landuse\_fre\_3** | **landcover\_fre\_2** | **landcover\_fre\_7** | **landcover\_fre\_5** | **landcover\_fre\_6** | **landcover\_fre\_1** | **landcover\_fre\_4** | **landcover\_fre\_3** | **landcover\_fre\_0** | **geometry** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0 | 290.759983 | 314.046661 | 0.0 | 204.314286 | 51.582413 | 506.0 | 356.0 | 38.0 | NaN | ... | NaN | 8646.0 | 7153.0 | 4278.0 | 1879.0 | 560.0 | NaN | NaN | NaN | POLYGON ((422430.905 4645394.671, 422280.905 4... |
| **1** | 1 | 290.760010 | 314.477295 | 0.0 | 205.464286 | 51.738358 | 516.0 | 384.0 | NaN | NaN | ... | NaN | 11055.0 | 6545.0 | 2257.0 | 2655.0 | NaN | NaN | NaN | NaN | POLYGON ((422430.905 4645544.671, 422280.905 4... |
| **2** | 2 | 290.759995 | 314.740021 | 0.0 | 203.657143 | 49.593369 | 531.0 | 160.0 | 209.0 | NaN | ... | NaN | 5966.0 | 8464.0 | 3730.0 | 4233.0 | 123.0 | NaN | NaN | NaN | POLYGON ((422430.905 4645694.671, 422280.905 4... |

3 rows × 71 columns

下面打印了处理后的各个数据，检查处理后的数据是否正确。

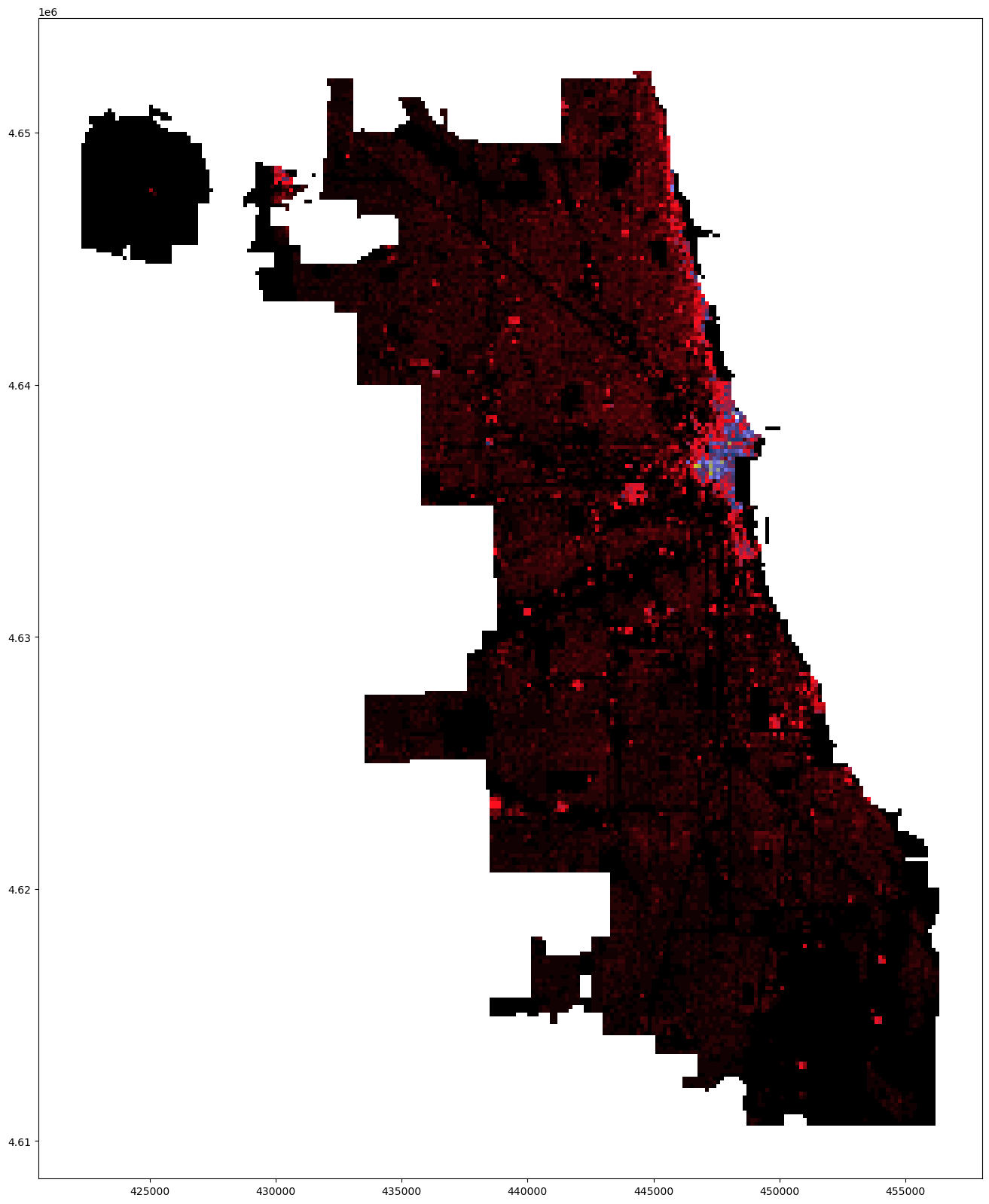
DEM数据检查。

zs\_gdf.plot(column='dem\_mean',legend=False,linewidth=0,figsize=(20,20),cmap='terrain');



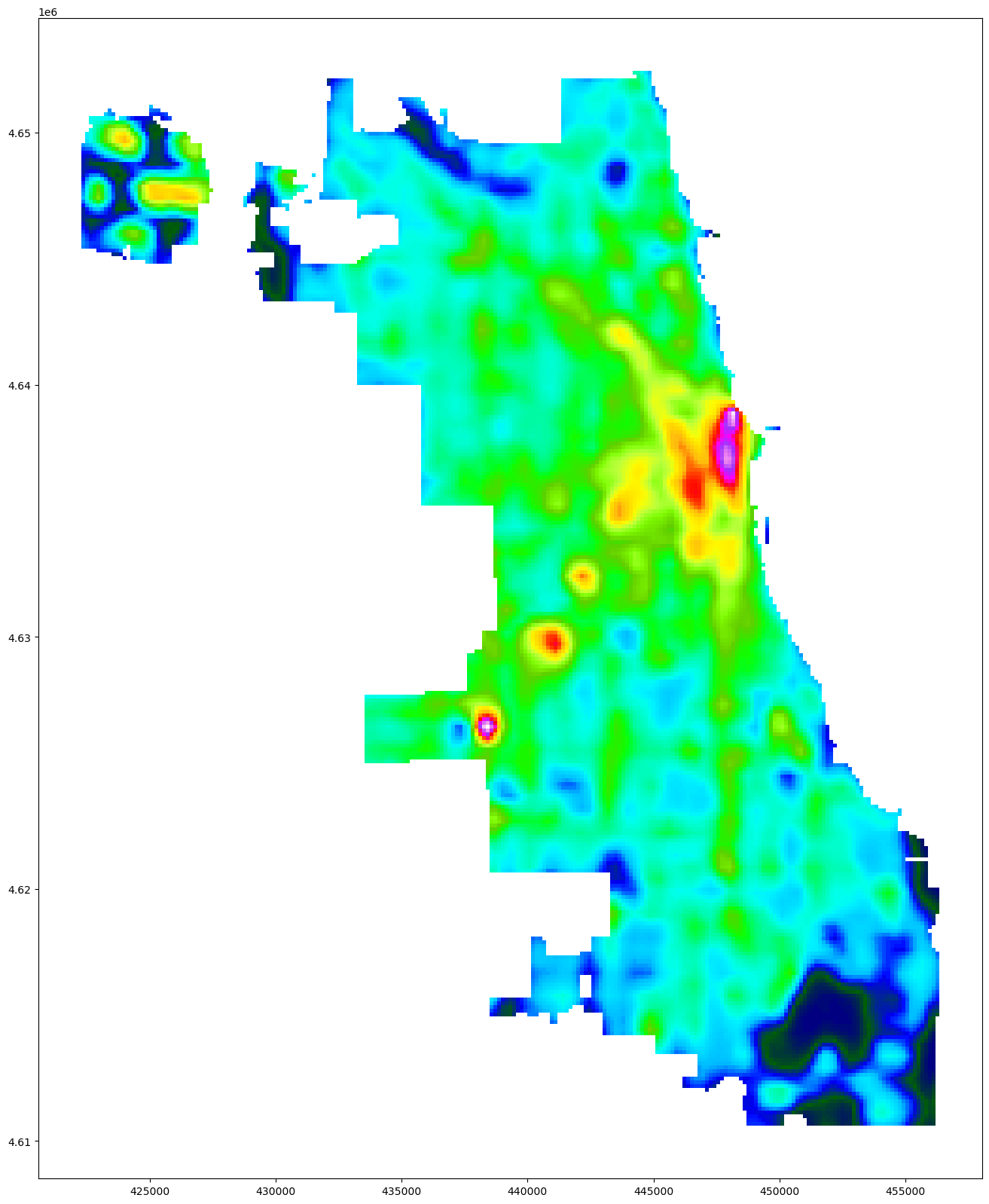
建筑高度数据检查。

zs\_gdf.plot(column='stories\_mean',legend=False,linewidth=0,figsize=(20,20),cmap='gist\_stern');



夜间灯光数据检查。

zs\_gdf.plot(column='nightlight\_mean',legend=False,figsize=(20,20),cmap='gist\_ncar');

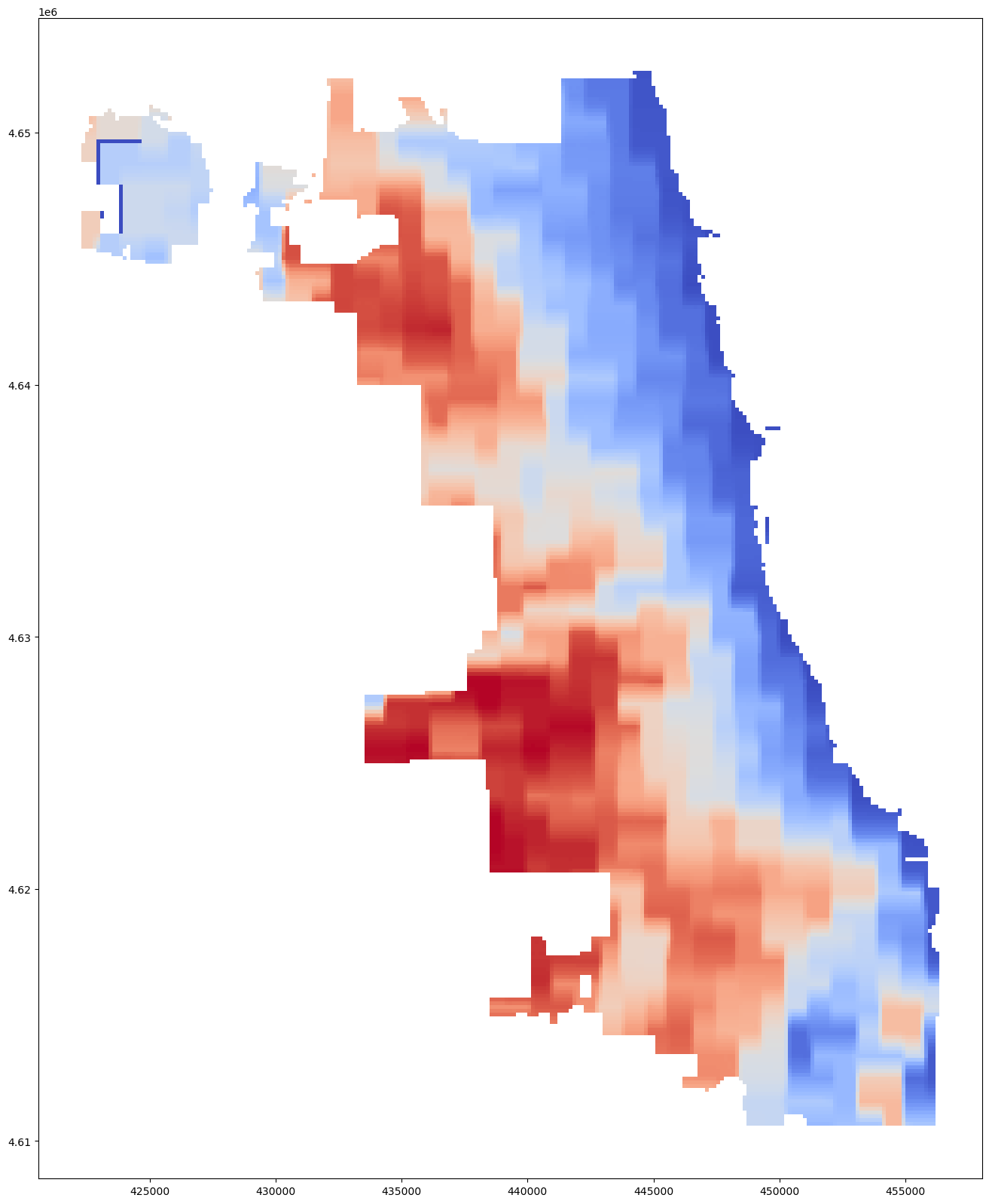


地表温度数据检查。对空值（数值为0的值）采用polynomial插值方法补全数据。

zs\_gdf['lst0226\_mean']=zs\_gdf['lst0226\_mean'].replace(0,None).interpolate(method='polynomial',order=5)

冬季地表温度数据检查。

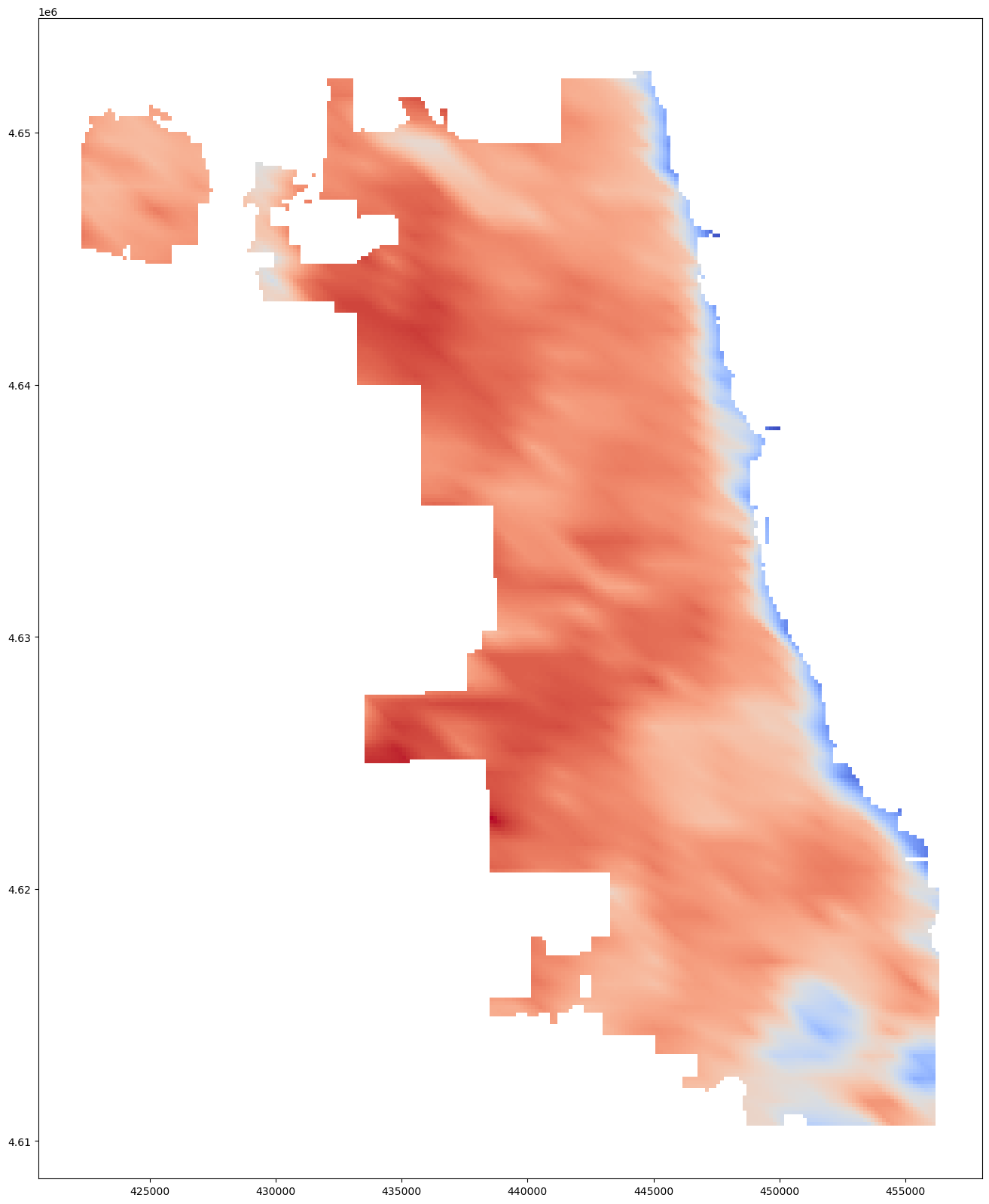
zs\_gdf.plot(column='lst0226\_mean',legend=False,figsize=(20,20),cmap='coolwarm');



夏季地表温度数据检查。

zs\_gdf['lst0810\_mean']=zs\_gdf['lst0810\_mean'].replace(0,None).interpolate(method='polynomial',order=5)

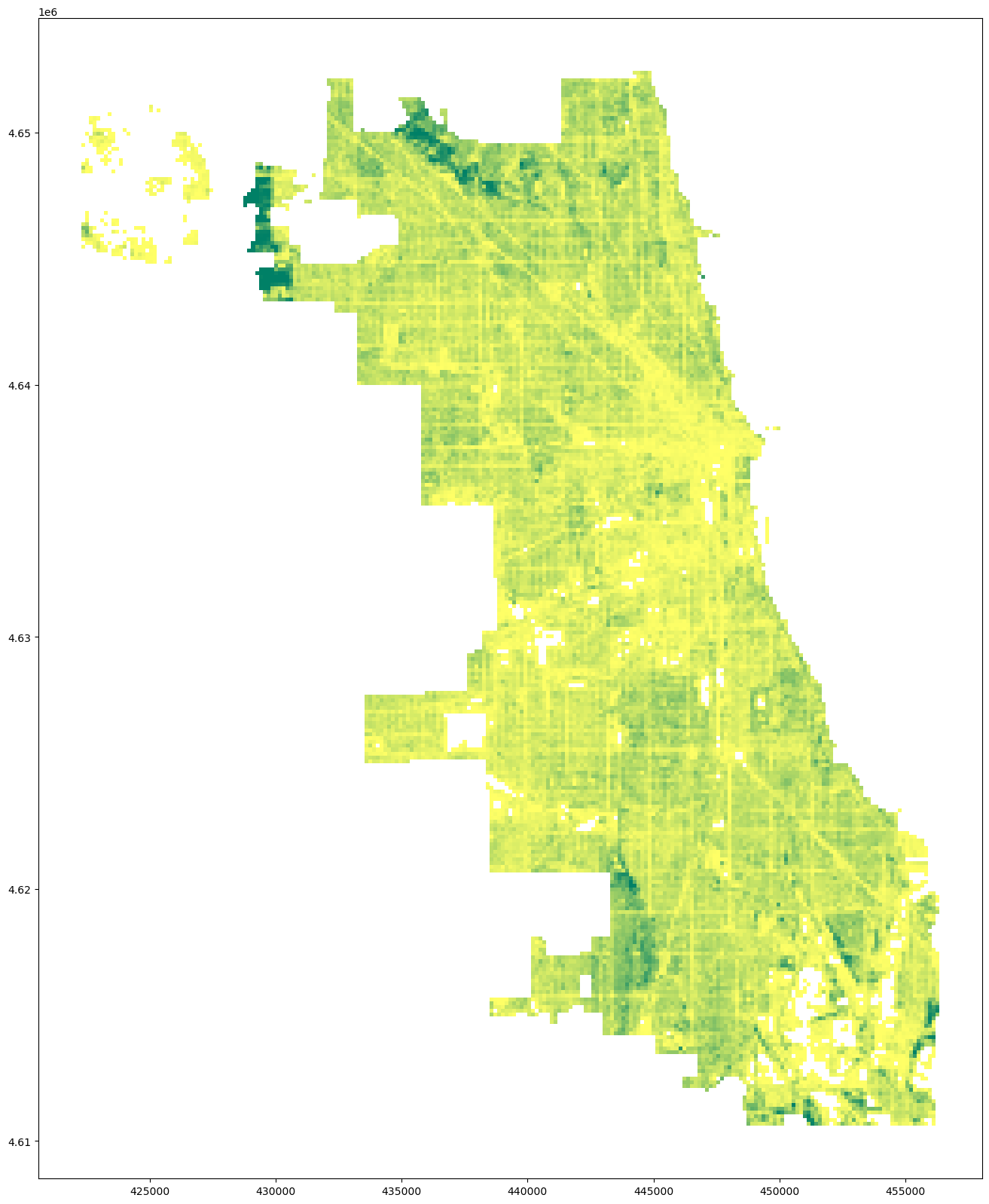
zs\_gdf.plot(column='lst0810\_mean',legend=False,figsize=(20,20),cmap='coolwarm');



土地覆盖类型数据检查。

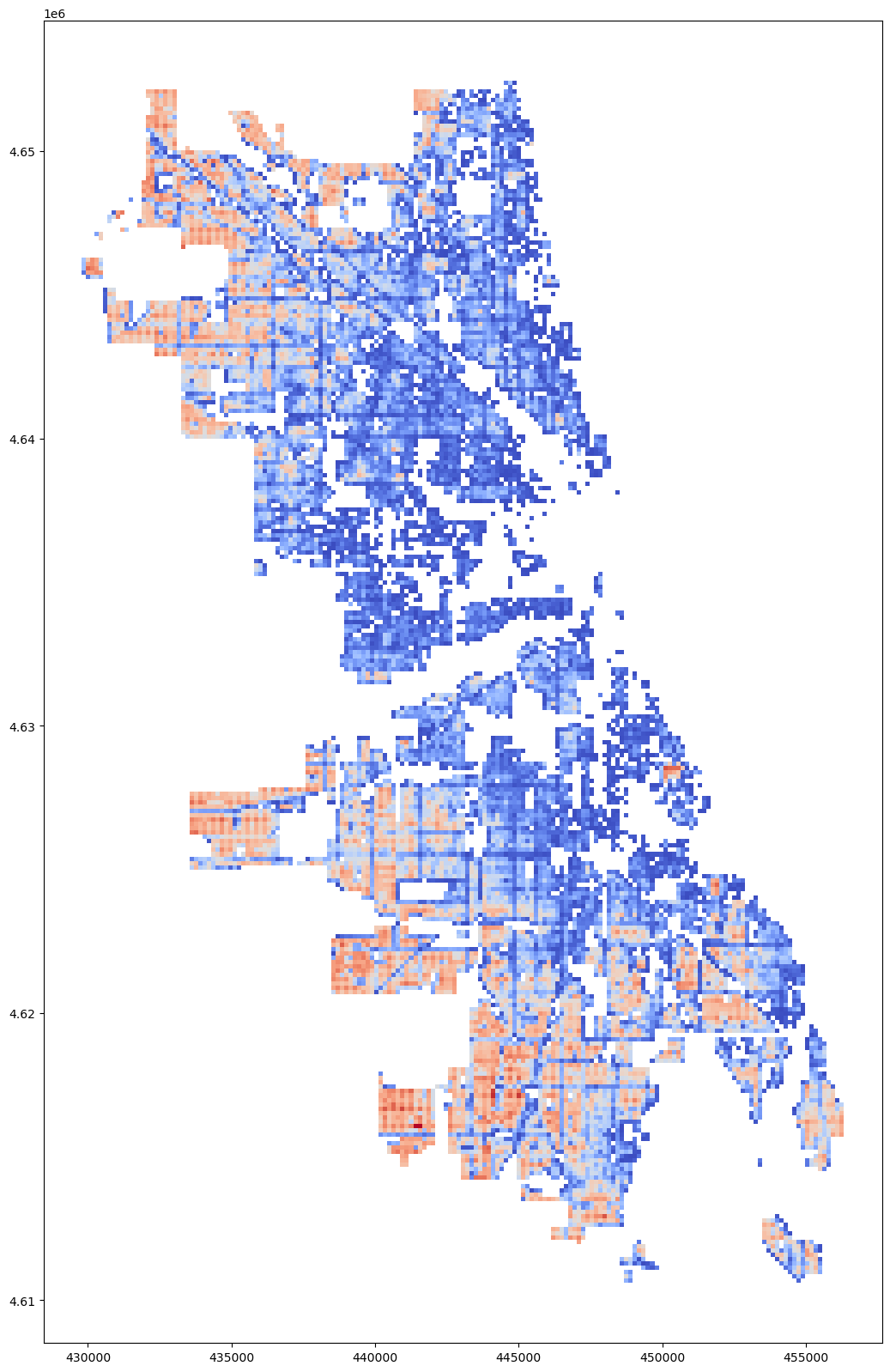
zs\_gdf.plot(column='landcover\_fre\_1',legend=False,figsize=(20,20),cmap=matplotlib.cm.get\_cmap('summer\_r'));

C:\Users\richi\AppData\Local\Temp\ipykernel\_30148\2541163892.py:1: MatplotlibDeprecationWarning: The get\_cmap function was deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get\_cmap(obj)`` instead.  
 zs\_gdf.plot(column='landcover\_fre\_1',legend=False,figsize=(20,20),cmap=matplotlib.cm.get\_cmap('summer\_r'));



土地利用类型数据检查。仅提取了索引值为0，即“独栋单户住宅”类型数据。

zs\_gdf.plot(column='landuse\_fre\_0',legend=False,figsize=(20,20),cmap='coolwarm');



* 标准化区域统计数据

根据不同数据类型采用不同的标准化方法，分为两类，一类是分类数据（土地利用和覆盖类型），采用各样方中各类型所占样方单元数的比例方式；另一类是连续值（地表温度、建筑高度、夜间灯光和DEM等），采用针对全部样方数据的最大最小值的归一化方法。

landuse\_cols=[i for i in zs\_gdf.columns if i.split('\_')[0]=='landuse']  
landcover\_cols=[i for i in zs\_gdf.columns if i.split('\_')[0]=='landcover']  
others\_cols=['lst0810\_mean','lst0226\_mean','stories\_mean','dem\_mean','nightlight\_mean']

zs\_norm\_gdf=zs\_gdf.copy(deep=True)  
zs\_norm\_gdf.fillna(0,inplace=True)  
zs\_norm\_gdf[landuse\_cols]=zs\_norm\_gdf.apply(lambda row:[row[i]/row[landuse\_cols].sum() for i in landuse\_cols],axis=1,result\_type="expand")  
zs\_norm\_gdf[landcover\_cols]=zs\_norm\_gdf.apply(lambda row:[row[i]/row[landcover\_cols].sum() for i in landcover\_cols],axis=1,result\_type="expand")  
zs\_norm\_gdf.loc[:,others\_cols]=zs\_norm\_gdf.loc[:,others\_cols].apply(lambda x: (x-x.min())/ (x.max()-x.min()), axis=0)  
zs\_norm\_gdf.to\_file(args.data.zs\_norm\_gdf\_fn,driver='GPKG')

zs\_norm\_gdf=gpd.read\_file(args.data.zs\_norm\_gdf\_fn)  
cols=landuse\_cols+landcover\_cols+others\_cols  
zs\_norm\_df=zs\_norm\_gdf[cols]  
zs\_norm\_df.head(3)

|  | **landuse\_fre\_31** | **landuse\_fre\_28** | **landuse\_fre\_55** | **landuse\_fre\_29** | **landuse\_fre\_10** | **landuse\_fre\_43** | **landuse\_fre\_54** | **landuse\_fre\_19** | **landuse\_fre\_42** | **landuse\_fre\_0** | **...** | **landcover\_fre\_6** | **landcover\_fre\_1** | **landcover\_fre\_4** | **landcover\_fre\_3** | **landcover\_fre\_0** | **lst0810\_mean** | **lst0226\_mean** | **stories\_mean** | **dem\_mean** | **nightlight\_mean** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.562222 | 0.395556 | 0.042222 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.083452 | 0.024871 | 0.0 | 0.0 | 0.0 | 0.783021 | 0.988374 | 0.0 | 0.705866 | 0.143921 |
| **1** | 0.573333 | 0.426667 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.117937 | 0.000000 | 0.0 | 0.0 | 0.0 | 0.802192 | 0.988374 | 0.0 | 0.726938 | 0.144447 |
| **2** | 0.590000 | 0.177778 | 0.232222 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | ... | 0.188000 | 0.005463 | 0.0 | 0.0 | 0.0 | 0.813889 | 0.988374 | 0.0 | 0.693825 | 0.137212 |

3 rows × 69 columns

### 3.4.1.2 聚类模式特征分析

6类多源数据可以根据因果归为两类，因包括土地利用、土地覆盖、建筑高度和DEM数据；果包括地表温度和夜间灯光。这里聚类因中的土地覆盖、建筑高度和DEM数据，分析聚类簇的模式组成和结构；并计算果中地表温度局部空间自相关系数的冷热点，通过夏季和冬季两个数据提取夏季降温和冬季保温的区域，统计该区域聚类簇的频数，尝试找到形成冬夏相对舒适区域的影响因素。

#### 1) 聚类模式组成结构

聚类的算法采用了MiniBatchKMeans，并通过SelectKBest方法计算各个因素对聚类结果的影响分数（贡献度）。MiniBatchKMeans聚类方式需要指定聚类簇数，簇数的不同可能因素的贡献度不同，因此定义clustering\_minibatchkmeans\_selectkbest\_ns()方法实现上述计算同时，可以指定簇数列表，返回不同簇的聚类簇和因素贡献度结果。

from sklearn.feature\_selection import f\_classif  
  
lu\_stories\_dem\_cols=landcover\_cols+['stories\_mean','dem\_mean',]   
ns=list(range(2,51))  
lu\_stories\_dem\_labels,lu\_stories\_dem\_best\_scores=usda\_model.clustering\_minibatchkmeans\_selectkbest\_ns(zs\_norm\_df,lu\_stories\_dem\_cols,ns,f\_classif)

100%|████████████████████████████████████████████████████████████████████████| 49/49 [00:20<00:00, 2.43it/s]

返回的贡献度为一个数组，将其转换为DataFrame格式，其行为影响因素（区域统计结果对象）；列为因素对应簇数的贡献度得分。

lu\_stories\_dem\_best\_scores\_df=pd.DataFrame(lu\_stories\_dem\_best\_scores,columns=lu\_stories\_dem\_cols,index=ns).T  
lu\_stories\_dem\_best\_scores\_df.head(3)

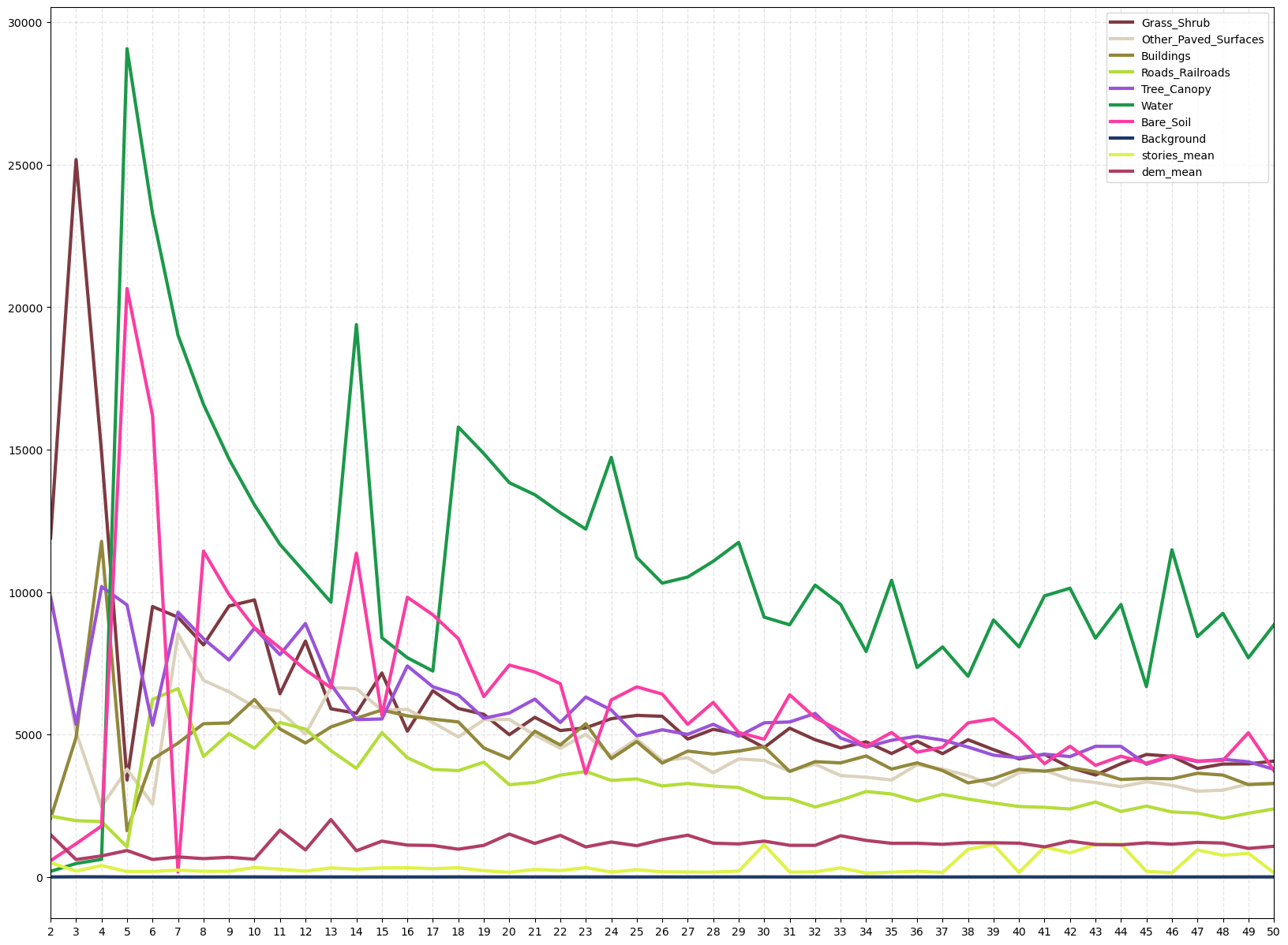
|  | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **...** | **41** | **42** | **43** | **44** | **45** | **46** | **47** | **48** | **49** | **50** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **landcover\_fre\_2** | 12170.457812 | 22459.522565 | 16231.941444 | 11790.434016 | 9118.860970 | 9429.347640 | 8557.231325 | 7214.913680 | 7831.697691 | 8716.563207 | ... | 4744.033372 | 4469.700813 | 4195.207935 | 3990.582413 | 4132.263827 | 4249.036397 | 3964.436883 | 4304.538001 | 3818.249952 | 3854.999734 |
| **landcover\_fre\_7** | 9475.753986 | 4988.439700 | 3516.760594 | 4461.817585 | 9013.292078 | 3485.742367 | 7245.912793 | 6365.899941 | 7775.607421 | 6488.888081 | ... | 3416.716437 | 3424.960984 | 3439.572117 | 3313.335208 | 3408.007615 | 3582.073340 | 3330.831808 | 3214.935771 | 3114.674266 | 3042.314222 |
| **landcover\_fre\_5** | 2575.552250 | 3793.440838 | 3093.660988 | 7101.933962 | 5759.085705 | 6024.705570 | 5963.282042 | 4453.227444 | 5019.705187 | 6700.553769 | ... | 3828.882111 | 3567.061152 | 3558.039171 | 3327.922843 | 3503.589146 | 3663.726931 | 3463.296281 | 3449.474086 | 3351.366060 | 3146.704440 |

3 rows × 49 columns

为了方便查看打印结果，将土地覆盖类型的索引值映射为对应的类型名称。下述打印的折线图横轴为簇数，纵轴为因素的贡献度得分。可以观察到簇数小于10左右时，因素贡献度的排序变化较大；大于簇数10左右时趋于稳定，其中水体对簇结果的影响较大；建筑高度、DEM和道路分类数据对结果的贡献度相对较小；其它土地覆盖类型贡献度相对集中。

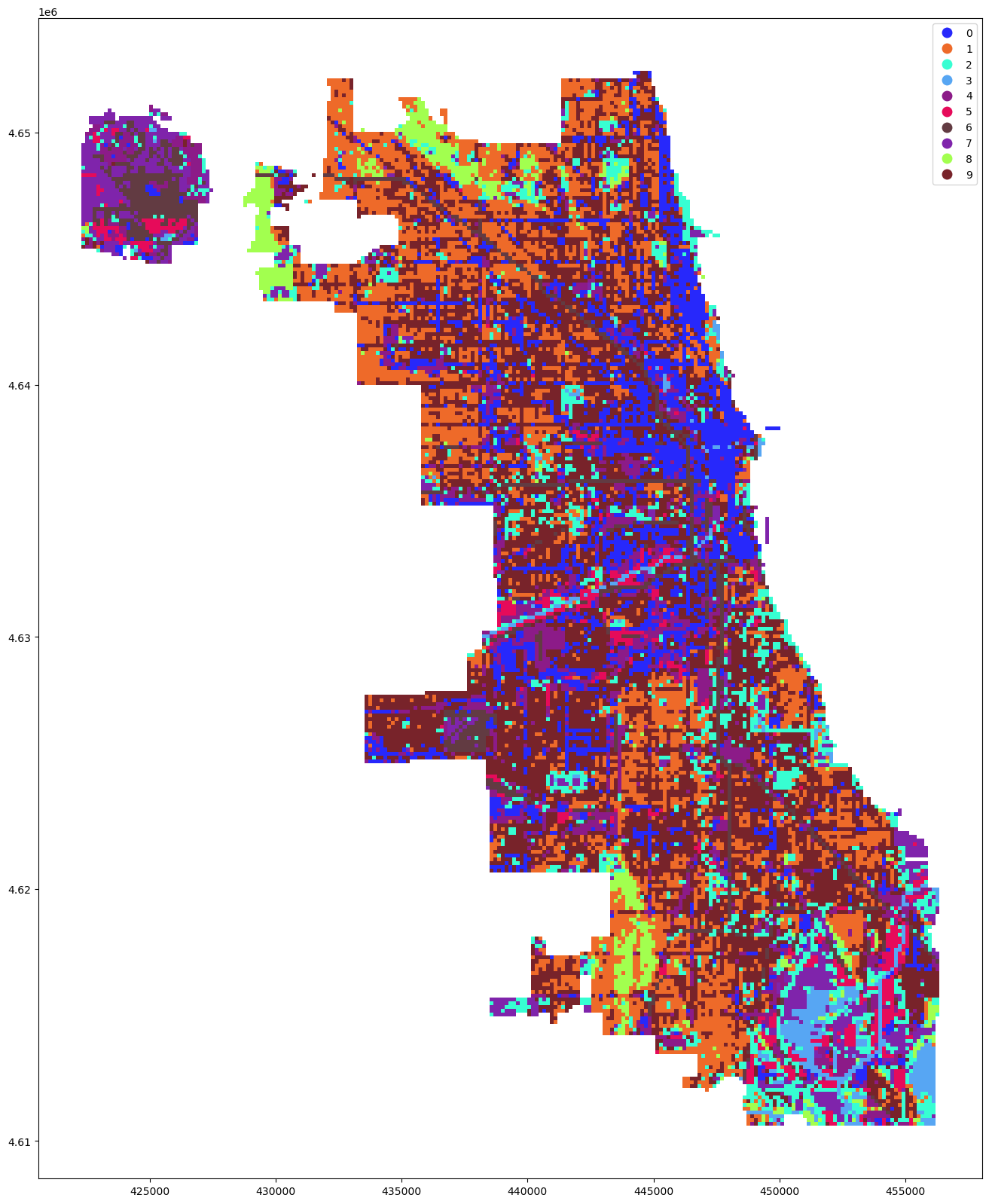
lc\_mapping={  
 'landcover\_fre\_0':'Background',  
 'landcover\_fre\_1':'Tree\_Canopy',  
 'landcover\_fre\_2':'Grass\_Shrub',  
 'landcover\_fre\_3':'Bare\_Soil',  
 'landcover\_fre\_4':'Water',   
 'landcover\_fre\_5':'Buildings',  
 'landcover\_fre\_6':'Roads\_Railroads',  
 'landcover\_fre\_7':'Other\_Paved\_Surfaces'}

fig, ax=plt.subplots(figsize=(20,15))  
  
np.random.seed(50)  
cmap=matplotlib.colors.ListedColormap (np.random.rand(256,3))  
lu\_stories\_dem\_best\_scores\_df.reset\_index(names=['lu'],inplace=True)  
lu\_stories\_dem\_best\_scores\_df.replace({'lu':lc\_mapping},inplace=True)  
pd.plotting.parallel\_coordinates(lu\_stories\_dem\_best\_scores\_df,'lu',ax=ax,colormap=cmap,axvlines=False,linewidth=3)  
ax.grid(color='grey', linestyle='--', linewidth=1,alpha=0.2)  
plt.show()



下述打印簇数为10的聚类结果地图，观察分布特征。

lu\_stories\_dem\_labels\_10=lu\_stories\_dem\_labels[10-2]  
clustering10\_lu\_stories\_dem\_gdf=zs\_norm\_gdf.copy(deep=True)  
clustering10\_lu\_stories\_dem\_gdf['label']=lu\_stories\_dem\_labels\_10  
  
np.random.seed(90)  
cmap=matplotlib.colors.ListedColormap (np.random.rand(256,3))  
  
clustering10\_lu\_stories\_dem\_gdf.plot(column='label',cmap=cmap,categorical=True,figsize=(20,20),legend=True);



分析聚类簇模式组成结构，通过计算簇中影响因素的频数比例进行分析。借鉴类/簇大小直方图样方标记特征，将频数分为3个组距，为(0.0, 0.3]、(0.3, 0.6]和(0.6, 1.0]。统计因素在各区间所占的比例，观察簇中因素的主要构成。从下述打印的各簇因素组距直方图可以得知，

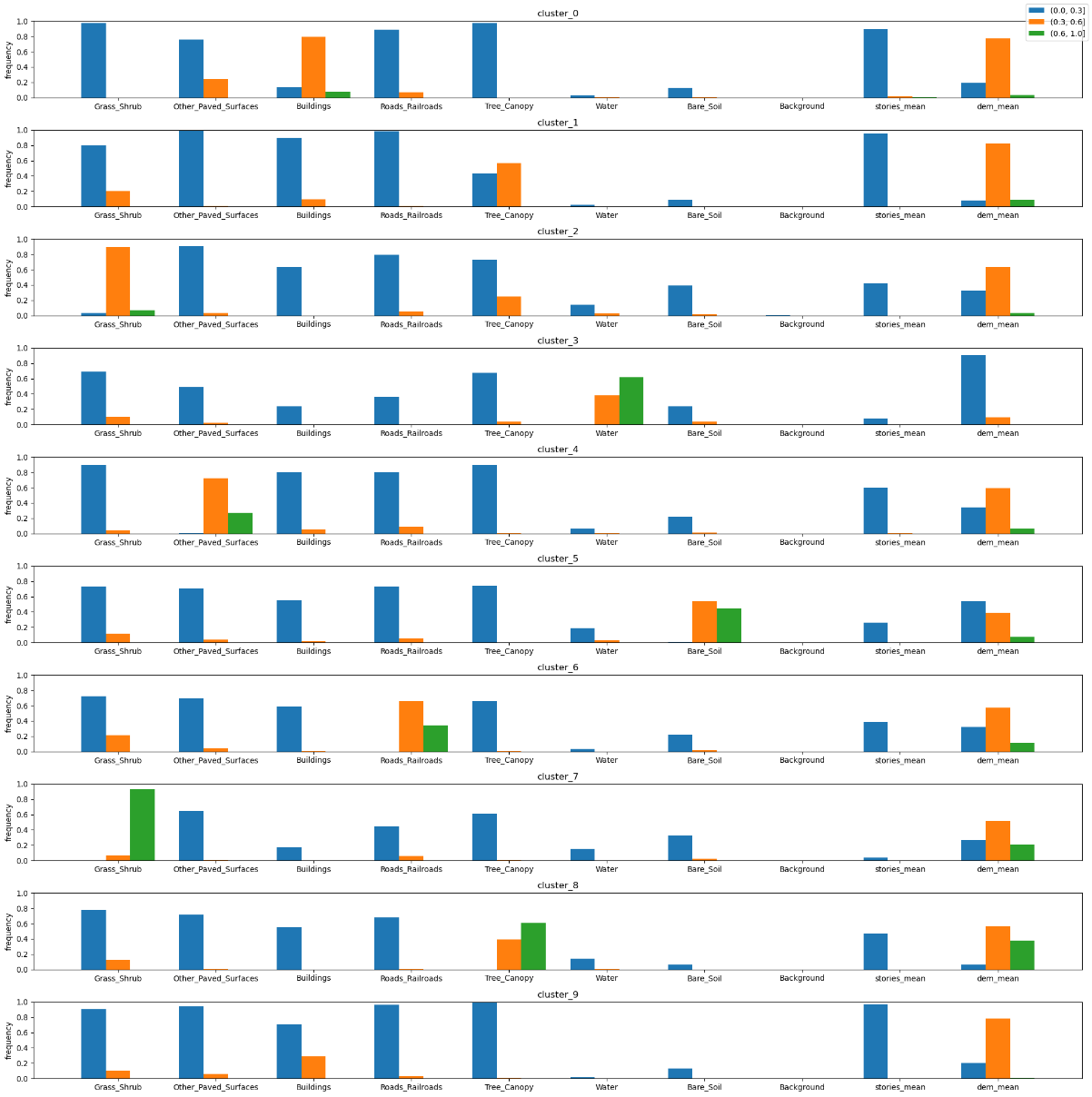
1. 簇0以建筑为主，其次为道路和铺地；
2. （不明显）簇1以林木、草地为主，其次为建筑；
3. 簇2以草地，林木为主，其次为建筑；
4. 簇3以水体为主要，其次为草地；
5. 簇4以铺地为主，其次为道路、建筑等；
6. 簇5以裸地为主，其次为草地；
7. 簇6以道路为主，其次为草地；
8. 簇7以草地为主，其次为道路；
9. 簇8以林木为主，其次为从草地；
10. （不明显）簇9以建筑为主，其次为草地。

上述标识为（不明显）的簇不含有(0.6, 1.0]区间值。通过因素组距直方图分析，可以找到簇数为10各簇较为明显的组成因素，及各因素的组成份额。

lu\_storites\_dem\_gbh\_signature=usda\_signature.group\_bins\_histogram(clustering10\_lu\_stories\_dem\_gdf[lu\_stories\_dem\_cols+['label']],lu\_stories\_dem\_cols+['label'],'label',bins=[0,0.3,0.6,1])  
lu\_storites\_dem\_gbh\_signature  
signature\_cluster\_0=lu\_storites\_dem\_gbh\_signature[0].rename(columns=lc\_mapping)  
signature\_cluster\_0

|  | **Grass\_Shrub** | **Other\_Paved\_Surfaces** | **Buildings** | **Roads\_Railroads** | **Tree\_Canopy** | **Water** | **Bare\_Soil** | **Background** | **stories\_mean** | **dem\_mean** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **(0.0, 0.3]** | 0.973634 | 0.755718 | 0.131512 | 0.889454 | 0.971728 | 0.028907 | 0.12770 | 0.0 | 0.891677 | 0.194409 |
| **(0.3, 0.6]** | 0.000318 | 0.237929 | 0.794473 | 0.065756 | 0.000000 | 0.002541 | 0.00413 | 0.0 | 0.016836 | 0.775731 |
| **(0.6, 1.0]** | 0.000000 | 0.000000 | 0.074015 | 0.000000 | 0.000000 | 0.000000 | 0.00000 | 0.0 | 0.002541 | 0.029860 |

fig, axs=plt.subplots(len(lu\_storites\_dem\_gbh\_signature.keys()), 1,figsize=(20,20)) # sharex=True,sharey=True,  
i=0  
for c,sig in lu\_storites\_dem\_gbh\_signature.items():  
 width=0.25  
 multiplier=0  
 x=np.arange(len(sig.columns)) # the label locations  
 for attribute, measurement in sig.iterrows():  
 offset=width\*multiplier  
 rects=axs[i].bar(x+offset, measurement, width, label=attribute)  
 ax.bar\_label(rects, padding=3)  
 multiplier+=1   
 axs[i].set\_title(f'cluster\_{c}')  
 axs[i].set\_ylabel('frequency')  
 axs[i].set\_xticks(x + width, signature\_cluster\_0.columns)  
 # axs[i].legend(loc='upper right', ncols=1)  
 axs[i].set\_ylim(0, 1)  
 i+=1  
   
handles, labels = axs[0].get\_legend\_handles\_labels()  
fig.legend(handles, labels, loc='upper right')   
fig.tight\_layout()   
plt.show()



计算各簇之间的距离，可以分析各簇相似度（或不相似度），距离算法采用Jensen-Shan方法。从下述图表可以观察到，簇0-簇9（0.278），簇1-簇9（0.340），簇2-簇1（0.673），簇3-簇5（0.943），簇4-簇9（0.700），簇5-簇9（0.738），簇6-簇9（0.877），簇7-簇2（0.771），簇8-簇1（0.580），簇9-簇0（0.278），是各簇对应的最相似的簇。其中簇9出现5次，一定程度上可以说明簇数选择上可以聚类为9簇更为合理，从上述折线图中也可以观察到为簇数9时，各因素的贡献度相对分开不聚拢。为了证实因素贡献度的集聚情况和离散程度，借鉴最近邻指数（Nearest Neighbor Index，NNI）（参考*更新策略*一章）计算方法，将其转换为一维度情况，定义nni\_1d()函数，并计算标准差（参考值），按最近邻指数排序簇数，可以得知簇数为9时排序第6（总共49个簇数）。

sig\_distance=[]  
for i,sig\_i in lu\_storites\_dem\_gbh\_signature.items():  
 temp=[]  
 for j,sig\_j in lu\_storites\_dem\_gbh\_signature.items():  
 sig\_d=usda\_signature.Distances(sig\_i.to\_numpy().flatten(),sig\_j.to\_numpy().flatten()).shannon()['Jensen-Shan']  
 temp.append(sig\_d)   
 sig\_distance.append(temp)  
sig\_distance\_matrix=np.array(sig\_distance)  
sig\_distance\_matrix\_df=pd.DataFrame(sig\_distance\_matrix,index=lu\_storites\_dem\_gbh\_signature.keys(),columns=lu\_storites\_dem\_gbh\_signature.keys())  
sig\_distance\_matrix\_df

|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.000000 | 0.795558 | 1.266796 | 1.382026 | 0.827412 | 1.025276 | 1.160620 | 1.467070 | 1.387805 | 0.278598 |
| **1** | 0.795558 | 0.000000 | 0.673072 | 1.417022 | 1.052836 | 0.936881 | 1.069788 | 1.449296 | 0.580868 | 0.340732 |
| **2** | 1.266796 | 0.673072 | 0.000000 | 1.203973 | 1.243503 | 0.956532 | 1.022521 | 0.771125 | 1.120203 | 0.803615 |
| **3** | 1.382026 | 1.417022 | 1.203973 | 0.000000 | 1.319932 | 0.943701 | 1.151882 | 1.242519 | 1.546553 | 1.175315 |
| **4** | 0.827412 | 1.052836 | 1.243503 | 1.319932 | 0.000000 | 1.014741 | 1.114610 | 1.530795 | 1.457485 | 0.700409 |
| **5** | 1.025276 | 0.936881 | 0.956532 | 0.943701 | 1.014741 | 0.000000 | 0.971058 | 1.154118 | 1.171058 | 0.738562 |
| **6** | 1.160620 | 1.069788 | 1.022521 | 1.151882 | 1.114610 | 0.971058 | 0.000000 | 1.206701 | 1.305287 | 0.877695 |
| **7** | 1.467070 | 1.449296 | 0.771125 | 1.242519 | 1.530795 | 1.154118 | 1.206701 | 0.000000 | 1.441653 | 1.327839 |
| **8** | 1.387805 | 0.580868 | 1.120203 | 1.546553 | 1.457485 | 1.171058 | 1.305287 | 1.441653 | 0.000000 | 1.058073 |
| **9** | 0.278598 | 0.340732 | 0.803615 | 1.175315 | 0.700409 | 0.738562 | 0.877695 | 1.327839 | 1.058073 | 0.000000 |

打印两两距离矩阵。

sig\_distance\_matrix\_df.style.background\_gradient(cmap ='viridis').set\_properties(\*\*{'font-size': '20px'})

|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0.000000 | 0.795558 | 1.266796 | 1.382026 | 0.827412 | 1.025276 | 1.160620 | 1.467070 | 1.387805 | 0.278598 |
| **1** | 0.795558 | 0.000000 | 0.673072 | 1.417022 | 1.052836 | 0.936881 | 1.069788 | 1.449296 | 0.580868 | 0.340732 |
| **2** | 1.266796 | 0.673072 | 0.000000 | 1.203973 | 1.243503 | 0.956532 | 1.022521 | 0.771125 | 1.120203 | 0.803615 |
| **3** | 1.382026 | 1.417022 | 1.203973 | 0.000000 | 1.319932 | 0.943701 | 1.151882 | 1.242519 | 1.546553 | 1.175315 |
| **4** | 0.827412 | 1.052836 | 1.243503 | 1.319932 | 0.000000 | 1.014741 | 1.114610 | 1.530795 | 1.457485 | 0.700409 |
| **5** | 1.025276 | 0.936881 | 0.956532 | 0.943701 | 1.014741 | 0.000000 | 0.971058 | 1.154118 | 1.171058 | 0.738562 |
| **6** | 1.160620 | 1.069788 | 1.022521 | 1.151882 | 1.114610 | 0.971058 | 0.000000 | 1.206701 | 1.305287 | 0.877695 |
| **7** | 1.467070 | 1.449296 | 0.771125 | 1.242519 | 1.530795 | 1.154118 | 1.206701 | 0.000000 | 1.441653 | 1.327839 |
| **8** | 1.387805 | 0.580868 | 1.120203 | 1.546553 | 1.457485 | 1.171058 | 1.305287 | 1.441653 | 0.000000 | 1.058073 |
| **9** | 0.278598 | 0.340732 | 0.803615 | 1.175315 | 0.700409 | 0.738562 | 0.877695 | 1.327839 | 1.058073 | 0.000000 |

计算1维度的最近邻指数。

lu\_stories\_dem\_best\_scores\_nni\_df=lu\_stories\_dem\_best\_scores\_df.copy(deep=True).T  
val\_names=lu\_stories\_dem\_best\_scores\_nni\_df.columns  
max\_val=lu\_stories\_dem\_best\_scores\_nni\_df.to\_numpy().max()  
lu\_stories\_dem\_best\_scores\_nni\_df['nni']=lu\_stories\_dem\_best\_scores\_nni\_df.apply(lambda row:usda\_network.nni\_1d(row[val\_names].tolist(),max\_val)[0],axis=1)  
lu\_stories\_dem\_best\_scores\_nni\_df['std']=lu\_stories\_dem\_best\_scores\_nni\_df.apply(lambda row:row[val\_names].std(),axis=1)  
lu\_stories\_dem\_best\_scores\_nni\_df.sort\_values(by=['nni','std'],ascending=False).head(10)

|  | **landcover\_fre\_2** | **landcover\_fre\_7** | **landcover\_fre\_5** | **landcover\_fre\_6** | **landcover\_fre\_1** | **landcover\_fre\_4** | **landcover\_fre\_3** | **landcover\_fre\_0** | **stories\_mean** | **dem\_mean** | **nni** | **std** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **4** | 16231.941444 | 3516.760594 | 3093.660988 | 989.125102 | 8734.821306 | 74.130180 | 26173.520524 | 2.799040 | 215.947590 | 682.967678 | 95.385941 | 8766.510256 |
| **3** | 22459.522565 | 4988.439700 | 3793.440838 | 1434.945622 | 11541.859972 | 135.938953 | 439.467173 | 5.870753 | 284.113732 | 853.990535 | 83.738066 | 7218.601958 |
| **6** | 9118.860970 | 9013.292078 | 5759.085705 | 3007.994768 | 7225.832439 | 23415.699430 | 1762.266628 | 2.006683 | 190.642039 | 718.323101 | 80.672665 | 7056.920076 |
| **7** | 9429.347640 | 3485.742367 | 6024.705570 | 1331.593805 | 10040.757827 | 19355.818289 | 13482.805925 | 0.325885 | 226.188558 | 754.925089 | 65.629799 | 6546.507263 |
| **16** | 6317.341453 | 4758.620227 | 5108.953287 | 3411.664533 | 7678.583321 | 18053.644870 | 9543.521979 | 1.568578 | 275.353404 | 1146.152946 | 64.154270 | 5382.480896 |
| **9** | 7214.913680 | 6365.899941 | 4453.227444 | 4908.185638 | 7460.832452 | 14521.325463 | 9844.067084 | 0.498452 | 182.778024 | 1732.738458 | 43.884935 | 4492.743137 |
| **8** | 8557.231325 | 7245.912793 | 5963.282042 | 5666.462124 | 7402.586293 | 16569.805873 | 11275.608840 | 0.385508 | 256.813604 | 604.669381 | 42.747869 | 5234.060065 |
| **20** | 5744.841661 | 4878.385927 | 4722.899831 | 4109.115905 | 5858.137077 | 14234.930618 | 7470.585373 | 0.802378 | 259.786855 | 1124.097743 | 42.652099 | 4160.576003 |
| **22** | 6112.810738 | 4578.179482 | 4682.451513 | 3396.653274 | 5951.361424 | 12747.017421 | 3853.346397 | 1.224067 | 254.889555 | 1429.874818 | 38.160046 | 3677.024369 |
| **23** | 5146.010926 | 4281.806473 | 5000.470527 | 3561.352320 | 6132.901170 | 12182.092363 | 6393.615917 | 1.651258 | 247.517775 | 1415.834755 | 37.920809 | 3569.211497 |

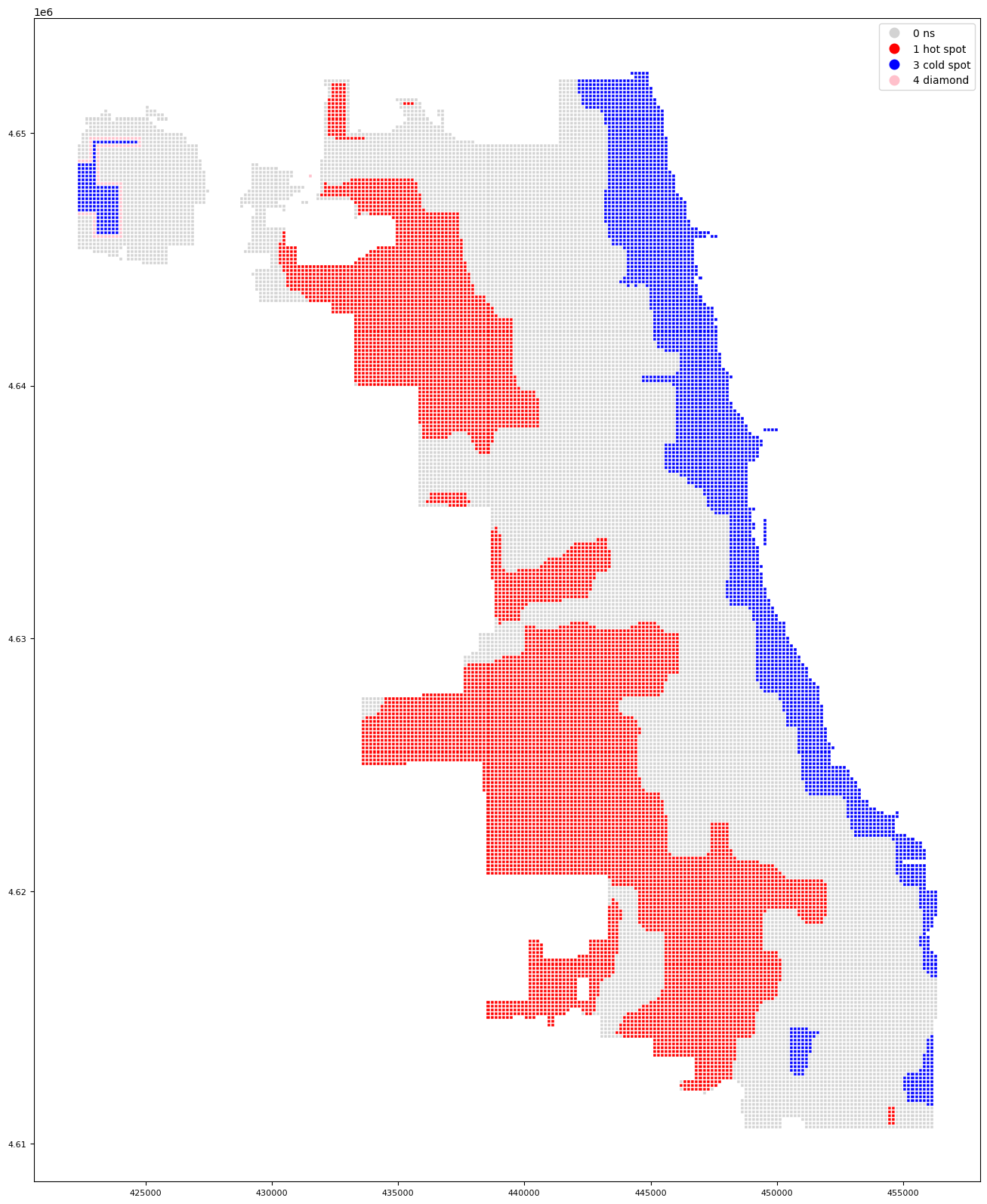
#### 2） 基于 LST 冷热点舒适区域与地表覆盖类型的簇分布统计

上述通过对“因”类数据聚类，分析了聚类簇的模式组成和结构。下述提取“果”的地表温度数据，计算局部空间自相关系数，提取夏季冷点和冬季非冷点的区域交集作为舒适区域。分析舒适区域簇数为10聚类结果各类簇的数量，分析簇与地表温度的关系。

首先计算冬季地表温度的局部空间自相关系数，提取统计显著性的冷热点。

la\_lst0226=usda\_model.moran\_local\_autocorrelation\_gdf(zs\_norm\_gdf,'lst0226\_mean')  
  
from matplotlib import colors  
cmap=colors.ListedColormap([ 'lightgrey', 'red', 'lightblue', 'blue', 'pink'])  
usda\_vis.gdf\_plot\_annotate(la\_lst0226,"cl\_li",annotate\_fontsize=10,categorical=True,cmap=cmap,figsize=(20,20))

('WARNING: ', 1505, ' is an island (no neighbors)')  
[1 1 1 ... 3 3 3]  
[0.01601994 0.01602003 0.01602001 ... 0.10347612 0.10290653 0.10158318]  
p\_value<0.05 num: 12179



计算夏季地表温度的局部空间自相关系数，提取统计显著性的冷热点。

la\_lst0810=usda\_model.moran\_local\_autocorrelation\_gdf(zs\_norm\_gdf,'lst0810\_mean')  
usda\_vis.gdf\_plot\_annotate(la\_lst0810,"cl\_li",annotate\_fontsize=10,categorical=True,cmap=cmap,figsize=(20,20))

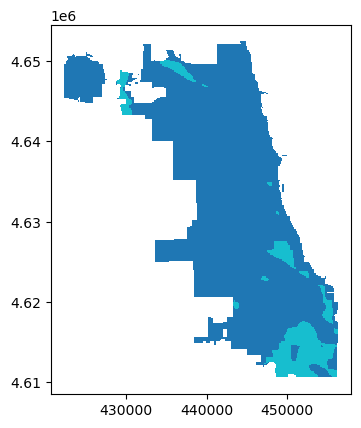
('WARNING: ', 1505, ' is an island (no neighbors)')  
[1 1 1 ... 3 3 3]  
[0.23452559 0.32599225 0.4316199 ... 4.88349186 4.03110304 4.6460232 ]  
p\_value<0.05 num: 11736

|  | **geometry** | **lst0810\_mean** | **li** | **p\_value\_li** | **li\_005** | **cl\_li** |
| --- | --- | --- | --- | --- | --- | --- |
| **0** | POLYGON ((422430.905 4645394.671, 422280.905 4... | 0.783021 | 1 | 0.192 | 0 | 0 ns |
| **1** | POLYGON ((422430.905 4645544.671, 422280.905 4... | 0.802192 | 1 | 0.108 | 0 | 0 ns |
| **2** | POLYGON ((422430.905 4645694.671, 422280.905 4... | 0.813889 | 1 | 0.075 | 0 | 0 ns |
| **3** | POLYGON ((422430.905 4645844.671, 422280.905 4... | 0.819604 | 1 | 0.070 | 0 | 0 ns |
| **4** | POLYGON ((422430.905 4645994.671, 422280.905 4... | 0.814433 | 1 | 0.096 | 0 | 0 ns |

计算夏季冷点和冬季非冷点的交集区域，为舒适区。

lst\_evaluation=la\_lst0810[['geometry']]  
lst\_evaluation['cozy']=np.array(la\_lst0810.cl\_li=='3 cold spot') & np.array(la\_lst0226.cl\_li!='3 cold spot')

lst\_evaluation.plot(column='cozy');



统计舒适区域簇的频数，得知簇2排序第1，其以草地和林地为主；其次为簇7、9、8；频数最小的为簇0和簇6，分布以建筑和道路为主要分布。上述结果与一般常识相符合。

lst\_evaluation['label']=clustering10\_lu\_stories\_dem\_gdf.label

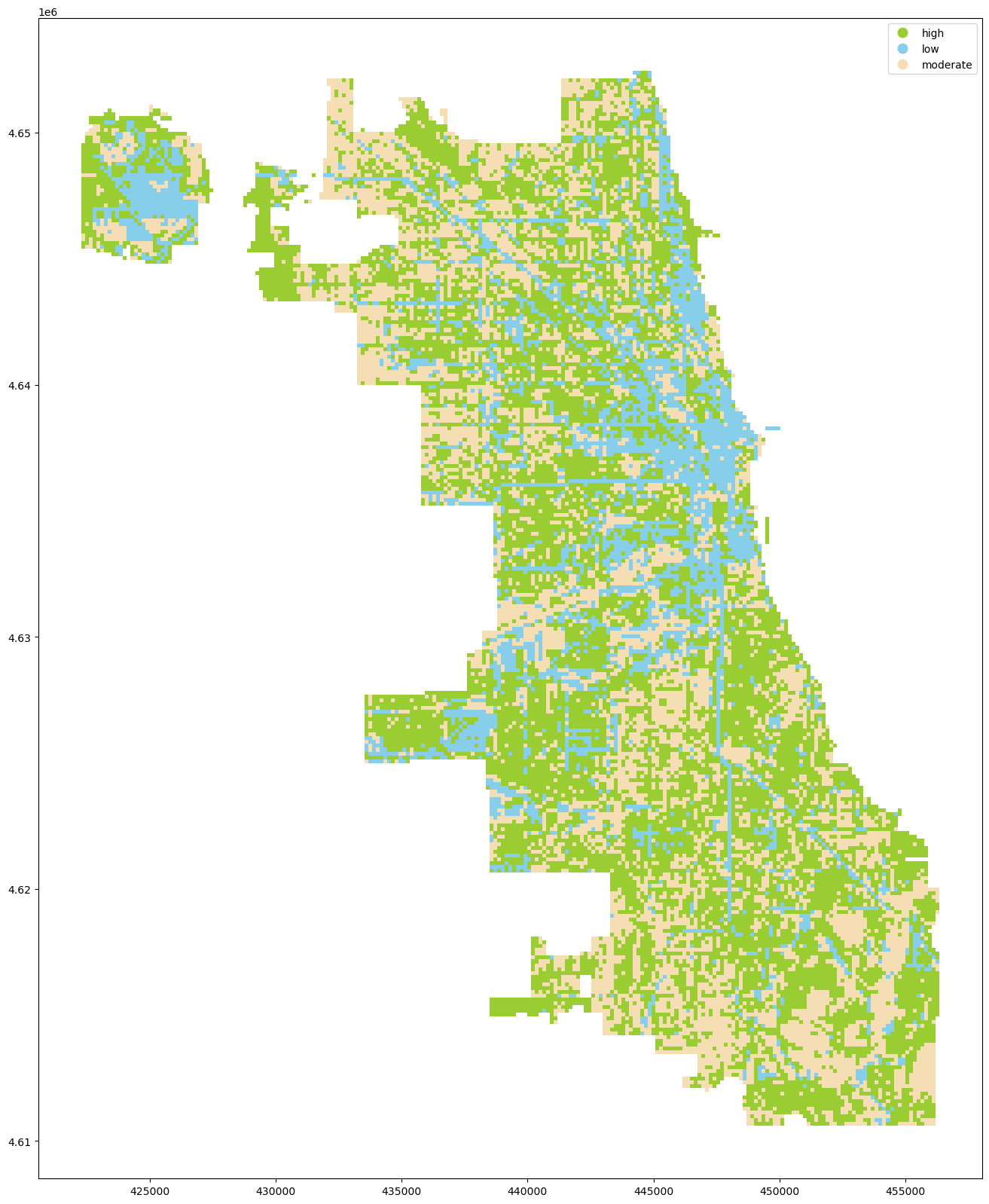
cozy\_labels=lst\_evaluation.label[lst\_evaluation.cozy==True]  
cozy\_clusters\_fre=cozy\_labels.value\_counts().sort\_values(ascending=False)  
cozy\_clusters\_fre

2 486  
7 351  
9 344  
8 343  
3 267  
1 223  
5 217  
4 170  
0 93  
6 59  
Name: label, dtype: int64

按照舒适区簇频数排序，分类为3个等级，‘low’、‘moderate’和’high’，绘制如下地图。’high’值区域相对具有较多草地和林地组成的样方，舒适度较好。

cozy\_clusters\_mapping={0:'low',1:'moderate',2:'high',3:'moderate',4:'moderate',5:'moderate',6:'low',7:'high',8:'high',9:'high'}  
lst\_evaluation['cozy\_rank']=lst\_evaluation.label.apply(lambda x:cozy\_clusters\_mapping[x])

cmap=colors.ListedColormap([ 'yellowgreen', 'skyblue', 'wheat'])  
lst\_evaluation.plot(column='cozy\_rank',cmap=cmap,categorical=True,figsize=(20,20),legend=True);



## 3.4.2 生成对抗网络（Generative Adversarial Networks，GAN） 的模式生成

生成对抗网络由 Ian J. Goodfellow 等人[1]于2014年提出，是一种深度神经网络架构，由一个生成网络（generative model，G）和一个判别网络（discriminative model，D）组成。通过 G 生成“假”数据，并试图欺骗 D 网络；而 D 网络对生成数据进行真伪鉴别，试图正确识别所有“假”数据。

生成对抗网络的提出，使得深度学习具有了“创造”的能力。GAN 目前已经发展到很多领域，例如字体生成、动漫角色生成、交互式图像生成、文字转图片、3D对象生成、图像编辑、面部年龄变化、人体姿态估计、风格样式迁移、高分辨率图像生成、物体检测识别、视频生成及预测、合成数据生成、实时人脸重建等[2]。上述应用中的风格样式迁移（style-transfer, pix2pix, sketch2image）、图像生成等，从一般意义的数据分析也跨度到了设计领域，为设计生成提供了深度学习网络模型原型。

就已有 GAN 网络模型的发展，结合城市空间数据分析内容，其中可以拓展的模式生成方式有，

1. 学习已有城市地理空间数据，例如土地利用和覆盖、遥感航拍影像等随机生成对应类型的数据（无条件约束）；
2. 使用 [Pix2pix](https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix)⑥条件对抗网络（Conditional Adversarial Networks）进行图像（图块模式，例如土地覆盖；或线条模式）到图像（例如遥感航拍影像）的转换[3]339（有条件约束）；
3. 修复或填充地理空间数据的空白区域[4]（有条件约束）；
4. 通过提高地理空间数据精度获得样方单元的生成信息[5]（有条件约束）；
5. DEM 表征三维空间二维数据的生成[6]（无条件约束）；
6. 结合参数化设计生成地理空间设计数据作为 GAN 学习样本，生成空间数据（无条件约束）。

### 3.4.2.1 NAIP航拍影像数据下载和制作样本数据集

#### 1) 数据下载

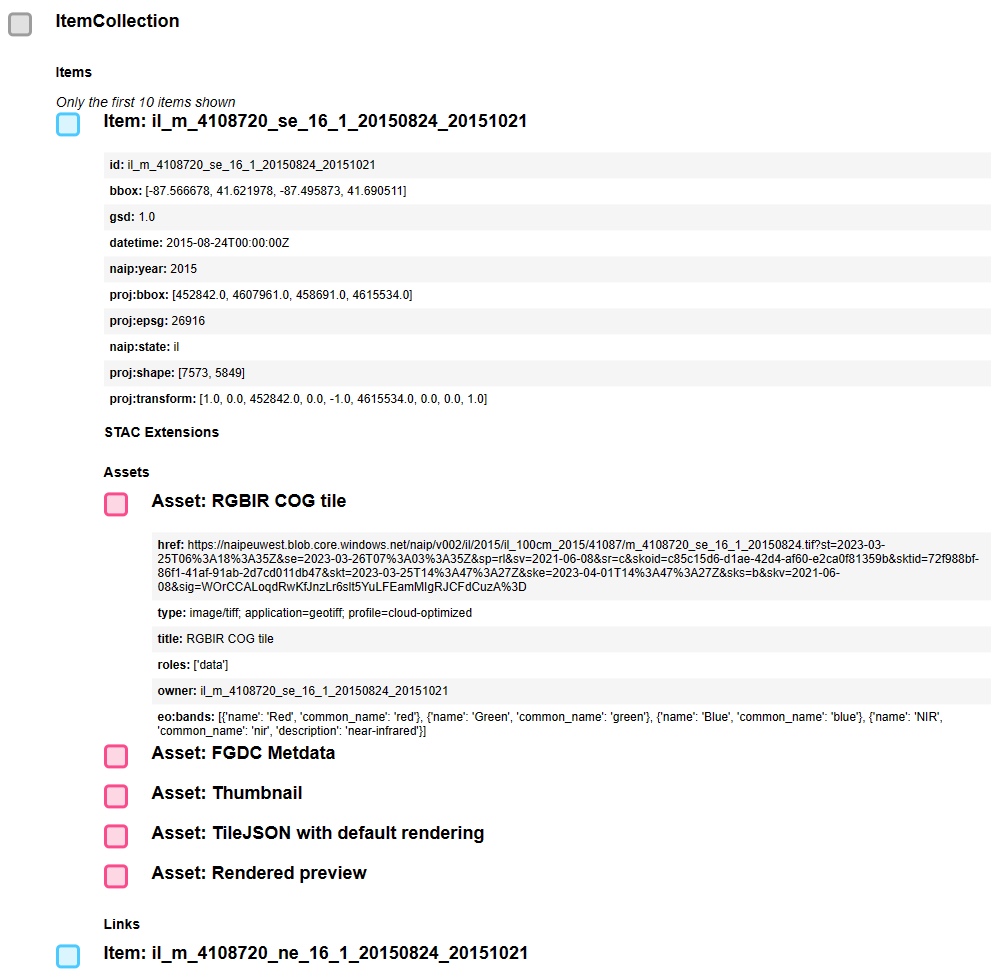
关于NAIP航拍影像数据的说明可以参考*NAIP航拍影像与分割模型库及Colaboratory和Planetary Computer Hub*一章。此次实验数据的下载方式参考[NAIP: National Agriculture Imagery Program](https://planetarycomputer.microsoft.com/dataset/naip#Example-Notebook)⑦提供的方法，通过给定边界boundary，指定日期范围range\_date="2015-01-01/2016-01-01"，直接下载数据。主要使用的核心库为planetary\_computer和pystac\_client。

import pathlib  
import rioxarray  
import pystac\_client  
import planetary\_computer  
from shapely.geometry import shape  
from IPython.display import Image  
import stackstac

搜索数据后，返回的items\_naip为pystac.item\_collection.ItemCollection对象，可以直接查看数据内容。

catalog = pystac\_client.Client.open(  
 "https://planetarycomputer.microsoft.com/api/stac/v1",  
 modifier=planetary\_computer.sign\_inplace,  
)  
  
boundary=gpd.read\_file(args.data.chicago\_boundary\_fn)  
  
area\_of\_interest = {  
 "type": "Polygon",  
 "coordinates": [list(boundary.geometry[0].envelope.exterior.coords)],  
 }  
  
range\_date="2015-01-01/2016-01-01"  
search\_naip=catalog.search(collections=["naip"], intersects=area\_of\_interest, datetime=range\_date)  
items\_naip=search\_naip.item\_collection()  
print(f"{len(items\_naip)} Items found in the date range")  
items\_naip

50 Items found in the date range



area\_of\_overlap函数可以移除空间重合的数据。例如不同年份时间段获得的影像通常是重合叠加的，一般搜索时配置为一个年份周期，避免重复和不同年份影像的拼接。

area\_shape = shape(area\_of\_interest)  
target\_area = area\_shape.area  
  
def area\_of\_overlap(item):  
 overlap\_area = shape(item.geometry).intersection(shape(area\_of\_interest)).area  
 return overlap\_area / target\_area  
  
items\_naip\_=sorted(items\_naip, key=area\_of\_overlap, reverse=True)  
print(len(items\_naip\_))  
items\_naip\_[:3]

50  
  
  
  
  
  
[<Item id=il\_m\_4108702\_sw\_16\_1\_20150822\_20151021>,  
 <Item id=il\_m\_4108702\_nw\_16\_1\_20150822\_20151021>,  
 <Item id=il\_m\_4108710\_nw\_16\_1\_20150822\_20151021>]

给定图像地址（url），可以直接显示查看图像。

Image(url=items\_naip\_[0].assets["rendered\_preview"].href)



可以不用传统意义上先下载数据后读取，而是直接读取打开图像数据，用于分析。

ds\_0=rioxarray.open\_rasterio(items\_naip\_[0].assets["image"].href).sel(band=[1, 2, 3])  
ds\_0

<xarray.DataArray (band: 3, y: 7593, x: 5854)>  
  
 [133348266 values with dtype=uint8]  
 Coordinates:  
 \* band (band) int32 1 2 3  
 \* x (x) float64 4.271e+05 4.271e+05 ... 4.329e+05 4.329e+05  
 \* y (y) float64 4.644e+06 4.644e+06 ... 4.636e+06 4.636e+06  
 spatial\_ref int32 0  
 Attributes:  
 AREA\_OR\_POINT: Area  
 scale\_factor: 1.0  
 add\_offset: 0.0

数据下载使用torchvision库提供的download\_url方法。下载前需要提取图像的网络地址（即字符?之前的内容）。

naip\_url\_lst=[item.assets['image'].href.split('?')[0] for item in items\_naip\_]  
from torchvision.datasets.utils import download\_url  
  
failed\_download\_info={}  
for idx,item\_url in enumerate(naip\_url\_lst):  
 print(idx,end=" ")  
 try:  
 download\_url(item\_url,args.data.naip\_root)  
 except:  
 failed\_download\_info[idx]=item\_url

0 Downloading https://naipeuwest.blob.core.windows.net/naip/v002/il/2015/il\_100cm\_2015/41087/m\_4108702\_sw\_16\_1\_20150822.tif to F:\data\NAIP\_chicago\m\_4108702\_sw\_16\_1\_20150822.tif  
  
  
100%|█████████████████████████████████████████████████████| 186139876/186139876 [01:31<00:00, 2028283.34it/s]  
  
  
1 Downloading https://naipeuwest.blob.core.windows.net/naip/v002/il/2015/il\_100cm\_2015/41087/m\_4108702\_nw\_16\_1\_20150822.tif to F:\data\NAIP\_chicago\m\_4108702\_nw\_16\_1\_20150822.tif  
  
  
100%|█████████████████████████████████████████████████████| 186908973/186908973 [01:00<00:00, 3096779.52it/s]  
  
  
...  
  
  
31 Downloading https://naipeuwest.blob.core.windows.net/naip/v002/il/2015/il\_100cm\_2015/41087/m\_4108719\_sw\_16\_1\_20150822.tif to F:\data\NAIP\_chicago\m\_4108719\_sw\_16\_1\_20150822.tif  
  
  
 64%|██████████████████████████████████▋ | 118751232/184675672 [09:56<05:31, 198995.93it/s]  
  
  
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49

#### 2）样本数据制作

可以使用torchgeo库构建数据集（dataset）和数据加载器（dataloader），用于 GAN 模型训练。这里则将其转换为包含 RGB 三个通道的 JPG 格式图像，且将其保存为不同大小的图像（同一图像的不同分辨率，使用resize方法），置于对应的文件夹中。图像大小包括4、8、16、32、64、128、256和512等8类。一方面选择适合的分辨率用于常规 GAN 模型的训练；另一方面用于实验 [StyleGAN3](https://github.com/NVlabs/stylegan3)8 模型[7]。数据处理部分代码的实现迁移于[MarsGAN](https://github.com/kheyer/MarsGAN/blob/master/StyleGAN.ipynb)⑨，结合torchgeo库完成。

1. 用torchgeo库构建大小为512的图像数据集和数据加载器，并存储为JPG格式图像

深度学习通常需要“海量”的样本，根据样本数量的需求，随机生成图像大小为 512 像素的20,000个样本。

from torchgeo.datasets import NAIP,stack\_samples   
from torchgeo.samplers import RandomGeoSampler  
from torch.utils.data import DataLoader  
from pathlib import Path  
import os  
import PIL   
  
from fastai import \*  
from fastai.vision import \*

batch\_size=200  
image\_size=512  
workers=8  
length=20000  
num\_epochs=length//batch\_size  
  
naip\_dataset=NAIP(args.data.naip\_root)  
sampler=RandomGeoSampler(naip\_dataset, size=image\_size, length=length)  
dataloader=DataLoader(naip\_dataset, sampler=sampler, collate\_fn=stack\_samples,batch\_size=batch\_size,num\_workers=workers)

NAIP影像有4个波段（R、G、B和NIR），保留R、G和B波段存储为 JPG 3通道图像。（PNG 格式的图像存储大小要高于 JPG）

from tqdm import tqdm  
import os  
p\_path=r'I:\data\NAIP4StyleGAN\naip\_512'  
  
for i, data in tqdm(enumerate(dataloader, 0),total=num\_epochs,position=0, leave=True):  
 img\_batch=data['image'].numpy()  
 for idx,img in enumerate(img\_batch):  
 plt.imsave(os.path.join(p\_path,f'{i}\_{idx}.jpg'),img[:3].transpose(1,2,0)/255)   
   
 if i==num\_epochs:break

100%|██████████████████████████████████████████████████████████████████████| 100/100 [13:12<00:00, 7.92s/it]

1. 建立不同图像大小的样本数据

建立对应不同图像大小的文件夹，用于存储对应大小的图像数据。

path=Path(r'I:\data\NAIP4StyleGAN')  
size=[4,8,16,32,64,128,256]  
for s in size:  
 subpath=os.path.join(path,f'patches\_{s}')  
 if not os.path.exists(subpath):  
 os.makedirs(subpath)

在后续的 GAN 实验中会使用到[fastai](https://docs.fast.ai/)库，方便数据集处理、模型构建和训练等内容。下面代码中部分方法调用于fastai库，例如open\_image等。使用resize方法调整图像大小，用Pillow（PIL）库的方法保存图像。

patch\_fnames=os.listdir(path/'naip\_512')  
  
for f, filename in enumerate(patch\_fnames):   
 if f%1000 == 0:  
 print(f)  
   
 ims = [open\_image(path/'naip\_512'/filename) for i in range(7)]  
   
 im1 = ims[0].resize((3,256,256))  
 im2 = ims[1].resize((3,128,128))   
 im3 = ims[2].resize((3,64,64))  
 im4 = ims[3].resize((3,32,32))  
 im5 = ims[4].resize((3,16,16))  
 im6 = ims[5].resize((3,8,8))  
 im7 = ims[6].resize((3,4,4))  
   
 PIL.Image.fromarray(image2np(im1.data\*255).astype(np.uint8)).save(path/'patches\_256'/filename, quality=95)  
 PIL.Image.fromarray(image2np(im2.data\*255).astype(np.uint8)).save(path/'patches\_128'/filename, quality=95)  
 PIL.Image.fromarray(image2np(im3.data\*255).astype(np.uint8)).save(path/'patches\_64'/filename, quality=95)  
 PIL.Image.fromarray(image2np(im4.data\*255).astype(np.uint8)).save(path/'patches\_32'/filename, quality=95)  
 PIL.Image.fromarray(image2np(im5.data\*255).astype(np.uint8)).save(path/'patches\_16'/filename, quality=95)  
 PIL.Image.fromarray(image2np(im6.data\*255).astype(np.uint8)).save(path/'patches\_8'/filename, quality=95)  
 PIL.Image.fromarray(image2np(im7.data\*255).astype(np.uint8)).save(path/'patches\_4'/filename, quality=95)

0  
1000  
2000  
3000  
4000  
5000  
6000  
7000  
8000  
9000  
10000  
11000  
12000  
13000  
14000  
15000  
16000  
17000  
18000  
19000

### 3.4.2.2 GAN 理解和fastai库 WGAN 实现——航拍影像生成实验

对 GAN 的实现综合使用[PyTorch](https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)⑩深度学习库和[fastai](https://docs.fast.ai/)⑪深度学习高阶 API（Applications）实现快速的数据处理和网络模型构建及训练。对 GAN 的解释主要来源于 [DCGAN TUTORIAL](https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html)⑫，[fastai-GAN](https://docs.fast.ai/vision.gan.html)⑬和*Generative Adversarial Nets*[1]，并混合解释了 GAN 网络中的 DCGANs（deep convolutional generative adversarial networks ）[8]和 WGAN (Wasserstein GAN)[9]，及相关的知识点。

构建用于深度学习的数据集（dataset）（含变换增强（Transforms，Augmentation））和数据加载器(dataloader)，通常使用基于PyTorch的扩展库[PyTorch Lightning](https://lightning.ai/docs/pytorch/stable/)⑭、[fastai](https://docs.fast.ai/)⑪、[TorchGeo](https://pytorch.org/blog/geospatial-deep-learning-with-torchgeo/)⑮等。下述使用了fastai构建的高阶 API。

from fastai.data.all import \*  
from fastai.callback.all import \*  
from fastai.vision.all import \*  
from fastai.vision.gan import \*  
  
from pathlib import Path  
import os

path=Path(r'I:\data\NAIP4StyleGAN')  
naip\_imgs\_path=path/'patches\_64' # Root directory for dataset  
ckpts\_path=Path('I:\model\_ckpts\naip\_GAN') # Root directory for model checkpoints

前文结合TorchGeo，使用 NAIP 航拍影像创建了 20,000个样本数据，通过get\_image\_files方法可以提取图像样本文件路径。

因为直接使用，例如 from fastai.data.all import \* 种方式调入库中方法函数，因此突然出现的一些方法通常为fastai库中的方法。

files=get\_image\_files(naip\_imgs\_path)  
print(files[:3])  
len(files)

[Path('I:/data/NAIP4StyleGAN/patches\_64/0\_0.jpg'), Path('I:/data/NAIP4StyleGAN/patches\_64/0\_1.jpg'), Path('I:/data/NAIP4StyleGAN/patches\_64/0\_10.jpg')]  
  
  
  
  
  
20000

使用DataBlock数据块方法快速构建数据集和数据加载器。

image\_size=64 # Spatial size of training images. All images will be resized to this size using a transformer.  
batch\_size=128 # Batch size during training  
nz=100 # Size of z latent vector (i.e. size of generator input)  
  
dblock=DataBlock(  
 blocks=(TransformBlock, ImageBlock),  
 get\_x=partial(generate\_noise,size=nz),  
 get\_items=get\_image\_files,  
 splitter=IndexSplitter([]),  
 item\_tfms=Resize(image\_size, method=ResizeMethod.Crop),   
 batch\_tfms=Normalize.from\_stats(torch.tensor([0.5,0.5,0.5]), torch.tensor([0.5,0.5,0.5])))  
  
dls=dblock.dataloaders(naip\_imgs\_path, path=naip\_imgs\_path, bs=batch\_size)

dls.show\_batch(max\_n=8)



GAN 是一个教授深度学习（Deep Learning，DL）模型捕获训练数据分布的一个框架。生成器（网络）不断尝试通过生成越来越好的假图像来“欺骗”判别器，判别器（网络）则努力鉴别，并对真假图像分类，这使得生成器生成的图像好像来自于训练数据真实的图像，而判别器总是以 50% 的置信度（confidence）猜测生成器生成的图像是真或假。

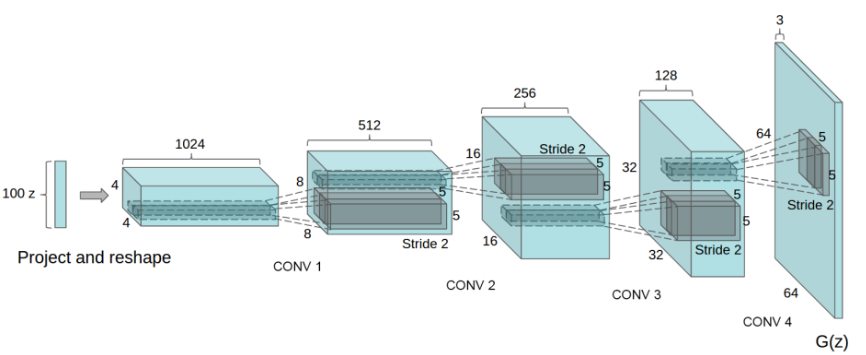
实验中使用转换 NAIP 的3通道（RGB）图像数据，因此配置通道（channel）数参数nc=3；图像大小使用 64 像素大小分辨率图像，因此配置生成器输出和判别器输入的图像大小参数ngf=ndf=64。

nc=3 # Number of channels in the training images. For color images this is 3  
ngf=64 # Size of feature maps in generator  
ndf=64 # Size of feature maps in discriminator

#### 1）生成器（G）的层结构

fastai集成了基本的生成网络，直接调用basic\_generator方法构建生成器，其中参数in\_sz为生成的噪声数据大小，配置为 100 维度（也为默认值），即从标准正态分布中采样的潜在空间向量（latent space vector），用表示。表示将映射到训练数据空间（data-space）的生成器函数，即变量generator。的目的是估计训练数据的分布获得生成器的估计分布来生成假数据。

打印generator，观察层结构，可以与下图[10]对照观察，但需要注意本次实验的生成网络第1个逆卷积（fractionally-strided convolution，deconvolution）[11]ConvTranspose2d的输出通道（out\_channels）为512，而图中从1024开始。



generator=basic\_generator(ngf, n\_channels=nc,in\_sz=100,n\_extra\_layers=1)  
generator

Sequential(  
 (0): AddChannels()  
 (1): ConvLayer(  
 (0): ConvTranspose2d(100, 512, kernel\_size=(4, 4), stride=(1, 1), bias=False)  
 (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): ReLU()  
 )  
 (2): ConvLayer(  
 (0): ConvTranspose2d(512, 256, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  
 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): ReLU()  
 )  
 (3): ConvLayer(  
 (0): ConvTranspose2d(256, 128, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  
 (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): ReLU()  
 )  
 (4): ConvLayer(  
 (0): ConvTranspose2d(128, 64, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  
 (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): ReLU()  
 )  
 (5): ConvLayer(  
 (0): ConvTranspose2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): ReLU()  
 )  
 (6): ConvTranspose2d(64, 3, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  
 (7): Tanh()  
)

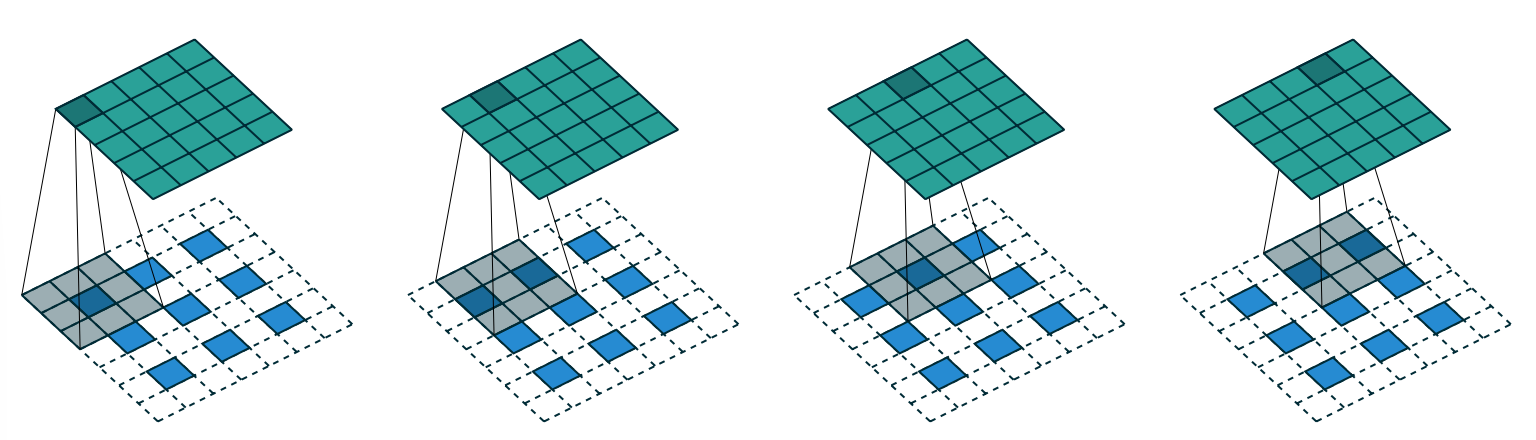
* 逆卷积层 —— ConvTranspose2d

逆卷积层可以对图像进行上采样，扩大图像的尺寸。逆卷积层数据输入和输出形状（shape）计算方法为：

当，输入（input）：或；输出（output）:或，式中为样本数，为图像（一个样本）的通道数，和为图像高和宽大小。

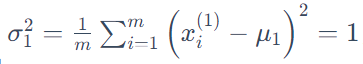
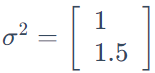
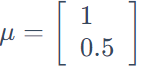
有，；。

逆卷积层可以看作相对于卷积Conv2d输入的梯度，计算原理可以从下图[12]理解。对的输入执行填充（padding）值0， 步幅（stride），大小卷积核卷积，为，；相当于，用大小卷积核卷积大小输入，卷积前输入值之间插值1个值0，执行填充值0，使用单位步幅，为。



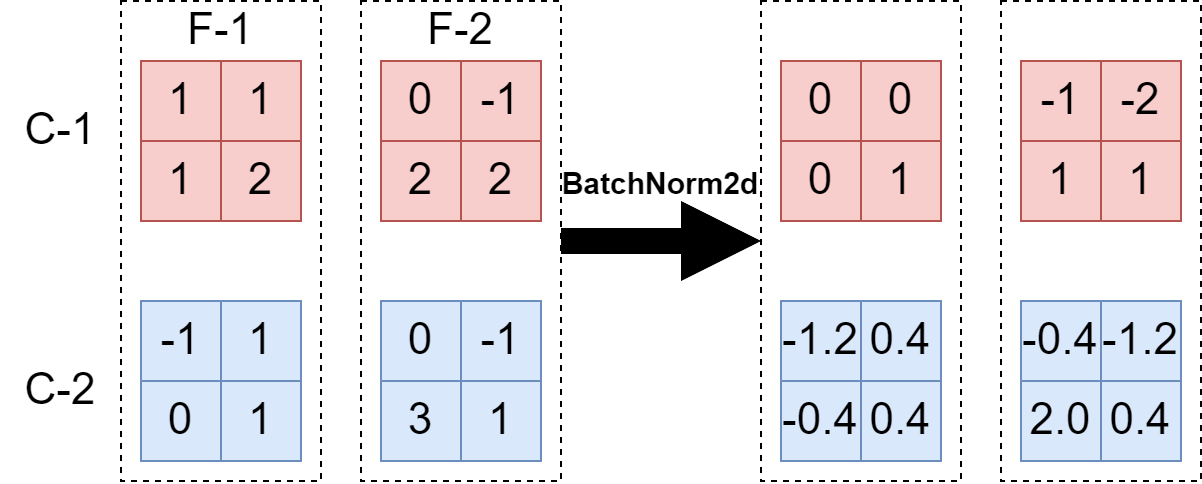
* 标准化层 —— BatchNorm2d

深度学习每一层输入因为前一层参数的改变，其分布在训练过程中都会发生变化，往往需要较小的学习率和和小心配置初始化参数，从而减慢了训练速度，尤其难以训练具有饱和非线性（例如使用sigmoid、tanh等激活函数）的模型，称这种现象为内部协变量偏移（internal covariate shift）。解决上述问题的方法是标准化（归一化）层输入，对每个训练小批量执行标准化，从而可以使用更高的学习率，且放宽初始化配置要求[13]。

如图，假设有两个图像的小批量数据样本，F-1和F-2，每个图像含2个通道C-1和C-2，则该小批量图像通道C-1的值集合表示为；图像通道C-2的值集合表示为。计算和的均值及方差有，， **

。

小批量标准化公式增加有两个调整参数，分别为和，默认情况下，，公式为：，或表示为：。



* 激活层 —— ReLU()和Tanh()

参考*从解析解到数值解，从机器学习到深度学习*一章中的激活函数部分。

下述实验代码是从数据集中提取一个样本，包括一个噪声数据和一个真实图像数据。对执行生成网络生成一个假图像，同时逐层执行，观察数据输入和输出的形状变化，即上采样过程。

可以直接根据图像路径，读取（Image.open）和打印（show\_image）一个真实图像im1。

im1=Image.open(files[1])  
ax=show\_image(im1, figsize=(2,2))  
print(im1.shape)

(64, 64)



通过dblock数据块读取数据，打印噪声数据和图像数据。

dsets=dblock.datasets(naip\_imgs\_path)  
im\_xy=dsets[1]  
im\_xy

(InvisibleTensor([-2.3454e+00, -1.5765e+00, 7.1060e-01, -5.7343e-01,  
 6.4001e-01, -6.0874e-01, -1.4418e-01, 4.7251e-01,  
 4.6599e-01, 3.3402e-01, -1.7018e+00, 3.8870e-01,  
 1.2323e-01, -2.1844e-01, 2.7379e-01, 3.6118e-01,  
 -1.2278e-01, -7.9387e-01, 5.2822e-02, -4.4322e-01,  
 1.5950e+00, 1.0422e+00, 1.0552e+00, -7.9903e-01,  
 -1.5224e+00, -4.7187e-01, -8.1115e-01, -3.8757e-01,  
 1.4381e+00, 5.6067e-01, -2.6452e-01, 1.5304e+00,  
 -1.3662e-01, 6.2670e-01, 7.5018e-01, 1.0567e+00,  
 -6.8847e-01, -1.6186e-01, -5.5967e-01, -2.2661e+00,  
 -3.4319e-01, -3.3552e-01, 1.1318e-01, 5.2318e-01,  
 1.0359e+00, -6.7418e-01, 5.8108e-01, 1.4617e+00,  
 1.0509e-01, -7.4633e-01, -2.3942e-01, -1.1583e+00,  
 2.5868e-01, -6.2029e-01, 1.3135e+00, -5.1513e-02,  
 -1.8016e+00, 5.2038e-01, -1.5848e-01, -1.1061e-01,  
 7.5131e-01, -6.9184e-01, -1.3670e-03, -4.6448e-01,  
 2.2224e+00, 7.4682e-01, 7.6649e-01, -3.5772e-01,  
 -7.1795e-02, -2.2193e+00, -9.1085e-01, -4.9485e-01,  
 5.5689e-01, 4.9285e-01, -1.7096e+00, 2.7568e+00,  
 -1.2357e+00, -2.5135e+00, -1.0657e+00, -1.0262e+00,  
 -1.1649e+00, -8.9837e-02, -1.1978e+00, 4.1369e-01,  
 2.3013e-01, -6.0128e-01, 1.4395e+00, -8.5148e-01,  
 -2.4028e-01, -1.6594e+00, 7.7949e-01, 8.2490e-01,  
 -8.5487e-01, -1.5042e+00, 2.2041e-01, -1.5568e+00,  
 5.8328e-01, -2.7559e-01, 8.5766e-01, 3.1389e-01]),  
 PILImage mode=RGB size=64x64)

提取噪声数据为im\_x；真实图像数据为im\_y。

im\_x=im\_xy[0]  
im\_y=im\_xy[1]  
print(im\_x.shape,im\_y.shape)  
print(im\_x[:10])

torch.Size([100]) (64, 64)  
InvisibleTensor([-2.3454, -1.5765, 0.7106, -0.5734, 0.6400, -0.6087, -0.1442,  
 0.4725, 0.4660, 0.3340])

ax=show\_image(im\_y, figsize=(2,2))



的第0层，AddChannels()，是将形状为(1,100)的噪声数据变换为(1,100,1,1)形状，用于卷积层的输入。

addChannels=generator[0]  
print(addChannels)  
im\_x\_Channels=addChannels(im\_x[None,:])  
print(im\_x\_Channels.shape)

AddChannels()  
torch.Size([1, 100, 1, 1])

提取第1个卷积层，可以通过属性查看参数等信息，例如权重值、填充等信息。

layer\_1=generator[1]  
print(layer\_1)  
print(layer\_1[1].weight)  
print(layer\_1[0].padding)

ConvLayer(  
 (0): ConvTranspose2d(100, 512, kernel\_size=(4, 4), stride=(1, 1), bias=False)  
 (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): ReLU()  
)  
Parameter containing:  
tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,  
 1., 1., 1., 1., 1., 1., 1., 1.], requires\_grad=True)  
(0, 0)

可以单独执行一个卷积层，查看比较数据及形状变化。

im\_x\_Channels\_layer1=layer\_1(im\_x\_Channels)  
im\_x\_Channels\_layer1.shape

torch.Size([1, 512, 4, 4])

分别单独执行生成器各层，查看比较数据形状变化，是如何由一个(1,100)的噪声数据，变换为(1,3,64,64)含3个通道，图像大小为的假图像数据。

im=im\_x[None,:]  
print(im.shape)  
for idx,layer in enumerate(generator):   
 im=layer(im)  
 print(f'{idx}------{im.shape}')

torch.Size([1, 100])  
0------torch.Size([1, 100, 1, 1])  
1------torch.Size([1, 512, 4, 4])  
2------torch.Size([1, 256, 8, 8])  
3------torch.Size([1, 128, 16, 16])  
4------torch.Size([1, 64, 32, 32])  
5------torch.Size([1, 64, 32, 32])  
6------torch.Size([1, 3, 64, 64])  
7------torch.Size([1, 3, 64, 64])

一次性执行generator，即生成器，获得假图像数据im\_x\_G。

im\_x\_G=generator(im\_x[None,:])  
print(im\_x\_G.shape)

torch.Size([1, 3, 64, 64])

噪声数据的大小可以调整，上述实验的噪声数据为100，下面配置为512，输出同样形状的假数据。

x=torch.randn(1, 512)  
generator(im\_x[None,:]).shape

torch.Size([1, 3, 64, 64])

#### 2） 判别器（D）的层结构

一个图像表示为，则为执行判别网络的结果，是鉴别来自于训练数据集的真实图像，还是由噪声通过生成的假图像的标量概率。如果来自于训练数据集，则值趋高；如果来自于生成器，则值趋低，因此可以认为是传统意义上的二元分类器（参考*逻辑回归二分类到 SoftMax 回归多分类*一章的解释）。对于来自的生成（假）图像的结果可表示为。试图鉴别假图像，表示为，而试图使生成的假图像趋真，即使的预测输出为假可能性降低，表示为。因此 GAN 的损失函数表示为： ，理论上极大极小博弈的解是当时。

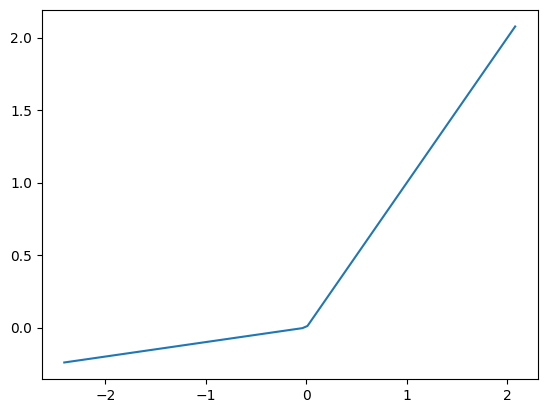
通过fastai的basic\_critic方法直接构建判别网络。与一样，逐层计算观察数据形状变化。网络卷积层包括Conv2d二维卷积，BatchNorm2d标准化，和激活函数nn.LeakyReLU。当使用 WGAN 网络时，的第6层配置为fastai.layers.Flatten(full=False)，仅将输入数据压成一维数据；当使用 DCGAN 时，最后一层为Sigmoid()层，将值归一化到[0,1]之间的一个标量概率，用于（真假）二元分类。

critic=basic\_critic(ndf, n\_channels=nc, n\_extra\_layers=1, act\_cls=partial(nn.LeakyReLU, negative\_slope=0.2))  
critic

Sequential(  
 (0): ConvLayer(  
 (0): Conv2d(3, 64, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1))  
 (1): LeakyReLU(negative\_slope=0.2)  
 )  
 (1): ConvLayer(  
 (0): Conv2d(64, 64, kernel\_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)  
 (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): LeakyReLU(negative\_slope=0.2)  
 )  
 (2): ConvLayer(  
 (0): Conv2d(64, 128, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  
 (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): LeakyReLU(negative\_slope=0.2)  
 )  
 (3): ConvLayer(  
 (0): Conv2d(128, 256, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  
 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): LeakyReLU(negative\_slope=0.2)  
 )  
 (4): ConvLayer(  
 (0): Conv2d(256, 512, kernel\_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)  
 (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track\_running\_stats=True)  
 (2): LeakyReLU(negative\_slope=0.2)  
 )  
 (5): Conv2d(512, 1, kernel\_size=(4, 4), stride=(1, 1))  
 (6): fastai.layers.Flatten(full=False)  
)

* LeakyReLU与ReLU主要区别是小于0的区间有一个倾斜，而不是完全水平的值0，如下图。

m=nn.LeakyReLU(0.1)  
input,\_=torch.randn(100).sort()  
output=m(input)  
  
fig=plt.figure()  
ax=fig.add\_subplot(1,1,1)  
ax.plot(input,output)  
plt.show()



逐层执行，查看数据和形状变化。输入为一个图像，形状为(1,3,64,64)，一个样本，含3个通道，大小为的图像；输出为一个标量值。

im=image2tensor(im\_y)[None,:]/255.  
print(im.shape)  
for idx,layer in enumerate(critic):   
 im=layer(im)  
 print(f'{idx}------{im.shape}')

torch.Size([1, 3, 64, 64])  
0------torch.Size([1, 64, 32, 32])  
1------torch.Size([1, 64, 32, 32])  
2------torch.Size([1, 128, 16, 16])  
3------torch.Size([1, 256, 8, 8])  
4------torch.Size([1, 512, 4, 4])  
5------torch.Size([1, 1, 1, 1])  
6------torch.Size([1, 1])

计算训练数据集真实图像的，并通过torch.sigmoid方法归一化。

critic.zero\_grad() # to make sure all grads are zero  
im\_y\_D=critic(image2tensor(im\_y)[None,:]/255.)  
print(im\_y\_D)  
im\_y\_D\_sigmoid=torch.sigmoid(im\_y\_D)  
print(im\_y\_D\_sigmoid)

tensor([[48.6151]], grad\_fn=<ViewBackward0>)  
tensor([[1.]], grad\_fn=<SigmoidBackward0>)

计算，即生成图像的，同样torch.sigmoid方法归一化。

im\_x\_G\_D=critic(im\_x\_G)  
print(im\_x\_G\_D)  
im\_x\_G\_D\_sigmoid=torch.sigmoid(im\_x\_G\_D)  
print(im\_x\_G\_D\_sigmoid)

InvisibleTensor([[39.8883]], grad\_fn=<AliasBackward0>)  
InvisibleTensor([[1.]], grad\_fn=<AliasBackward0>)

为了比较训练图像和生成假图像的结果，另从训练数据中又提取了一个真实图像，以两个真实图像和生成假图像比较。

im\_xy\_comp=dsets[7]  
im\_y\_comp=im\_xy\_comp[1]  
ax=show\_image(im\_y\_comp, figsize=(2,2))  
  
im\_y\_comp\_D=critic(image2tensor(im\_y\_comp)[None,:]/255.)  
print(im\_y\_comp\_D)  
im\_y\_comp\_D\_sigmoid=torch.sigmoid(im\_y\_comp\_D)  
print(im\_y\_comp\_D\_sigmoid)

tensor([[0.9176]], grad\_fn=<ViewBackward0>)  
tensor([[0.7146]], grad\_fn=<SigmoidBackward0>)



#### 3) 训练判别器

使用适用于0/1二分类的二值交叉熵损失（Binary Cross Entropy Loss，BCELoss）计算到真实标签（real label）值1，或假标签（fake labe）值0的距离（概率），其公式为：，式中，为真假标签（1或0），为预测输出（即）。BCELoss 的学习目标就是让逼近指定的标签值0或1，因此计算训练数据真实图像指定BCELossFlat()方法输入参数target为1；计算生成假图像指定BCELossFlat()方法输入参数target为0，训练模型。下述单样本计算的结果显示，训练数据真实值的 BCELoss 结果为1.047；生成假图像的 BCELoss 结果为0.143；参照图像结果为 0.336。对于生成假图像的计算如果指定的标签为真值1，则公式为；如果指定的标签为假值0，则公式为，因此下述代码tst(im\_x\_G\_D\_sigmoid,label\_fake)和tst(1-im\_x\_G\_D\_sigmoid,label\_real)计算结果同。

tst=BCELossFlat() # 相当于 nn.BCELoss()  
  
label\_real=torch.full((im\_y\_D\_sigmoid.size(0),), 1, dtype=torch.float)  
label\_fake=torch.full((im\_y\_D\_sigmoid.size(0),), 0, dtype=torch.float)  
print(f'real label:{label\_real};\nfake label:{label\_fake}')  
  
im\_y\_D\_sigmoid\_tst\_real=tst(im\_y\_D\_sigmoid,label\_real)  
im\_x\_G\_D\_sigmoid\_tst\_fake=tst(im\_x\_G\_D\_sigmoid,label\_fake) # 同 tst(1-im\_x\_G\_D\_sigmoid,label\_real)  
im\_y\_comp\_D\_sigmoid\_real=tst(im\_y\_comp\_D\_sigmoid,label\_real)  
print(f'im\_y\_D\_sigmoid\_tst\_real:\t{im\_y\_D\_sigmoid\_tst\_real}\nim\_x\_G\_D\_sigmoid\_tst\_fake:\t{im\_x\_G\_D\_sigmoid\_tst\_fake}\nim\_y\_comp\_D\_sigmoid\_real:\t{im\_y\_comp\_D\_sigmoid\_real}')

real label:tensor([1.]);  
fake label:tensor([0.])  
im\_y\_D\_sigmoid\_tst\_real: TensorBase(1.0470, grad\_fn=<AliasBackward0>)  
im\_x\_G\_D\_sigmoid\_tst\_fake: TensorBase(0.1433, grad\_fn=<AliasBackward0>)  
im\_y\_comp\_D\_sigmoid\_real: TensorBase(0.3361, grad\_fn=<AliasBackward0>)

训练的目标就是最大化给定输入（训练数据集真实图像和生成假图像）为真或假的概率，即最大化errD（）的概率。通过对，为训练数据真实图像，和，分别对应的变量为im\_y\_D\_sigmoid\_tst\_real和im\_x\_G\_D\_sigmoid\_tst\_fake，对其执行反向传播.backward后，结合优化算法更新网络的参数。优化算法对于 DCGAN 参考选择Adam，对于 WGAN 参考选择 RMSProp。下述针对 DCGAN 使用了Adam优化算法。

errD=im\_y\_D\_sigmoid\_tst\_real+im\_x\_G\_D\_sigmoid\_tst\_fake  
print(errD)

TensorBase(1.1903, grad\_fn=<AliasBackward0>)

查看网络的参数（权重（weights）和偏置（biases））。

list(critic.parameters())[0][:1]

tensor([[[[-0.0611, -0.2790, 0.1526, 0.0098],  
 [-0.2067, 0.0547, 0.0916, 0.1760],  
 [ 0.2402, 0.2054, -0.0340, 0.0343],  
 [ 0.0649, 0.2564, 0.2802, -0.0319]],  
  
 [[-0.1696, -0.1078, 0.3393, -0.2552],  
 [ 0.1721, 0.2788, 0.1575, 0.2830],  
 [-0.3075, 0.3486, 0.0972, 0.2338],  
 [-0.2867, 0.0078, -0.0618, -0.1351]],  
  
 [[-0.0362, 0.0999, 0.3280, 0.0710],  
 [-0.1982, 0.1575, 0.1228, 0.0044],  
 [ 0.1492, -0.1891, 0.0745, -0.2384],  
 [ 0.3277, 0.0755, -0.2309, -0.1302]]]], grad\_fn=<SliceBackward0>)

如果不执行反向传播，并不会更新模型的参数值，如下查看参数结果同上。

lr=0.1 # 2e-4  
opt\_D=Adam(critic.parameters(), lr=lr)  
opt\_D.step()  
list(critic.parameters())[0][:1]

tensor([[[[-0.0611, -0.2790, 0.1526, 0.0098],  
 [-0.2067, 0.0547, 0.0916, 0.1760],  
 [ 0.2402, 0.2054, -0.0340, 0.0343],  
 [ 0.0649, 0.2564, 0.2802, -0.0319]],  
  
 [[-0.1696, -0.1078, 0.3393, -0.2552],  
 [ 0.1721, 0.2788, 0.1575, 0.2830],  
 [-0.3075, 0.3486, 0.0972, 0.2338],  
 [-0.2867, 0.0078, -0.0618, -0.1351]],  
  
 [[-0.0362, 0.0999, 0.3280, 0.0710],  
 [-0.1982, 0.1575, 0.1228, 0.0044],  
 [ 0.1492, -0.1891, 0.0745, -0.2384],  
 [ 0.3277, 0.0755, -0.2309, -0.1302]]]], grad\_fn=<SliceBackward0>)

通过损失函数（BCELoss）计算预测值（）和对应标签（0/1）的误差后，当对误差张量（error tensor）调用.backward时，则会执行反向传播，然后自动梯度（Autograd）计算每个模型参数的梯度（gradients），并将其存储在参数的.grad属性中，且通过优化算法更新模型参数值。对网络模型（或）定义优化算法（优化器（optimizer）），调用optimizer.step()启动梯度下降，优化器通过存储在.grad属性中的梯度调整每个模型参数。

im\_y\_D\_sigmoid\_tst\_real.backward()  
im\_x\_G\_D\_sigmoid\_tst\_fake.backward()

opt\_D.step()  
list(critic.parameters())[0][:1]

tensor([[[[ 0.0387, -0.1787, 0.0524, -0.0902],  
 [-0.1065, 0.1546, -0.0084, 0.0758],  
 [ 0.1403, 0.3051, 0.0661, -0.0657],  
 [-0.0352, 0.3561, 0.1799, -0.1318]],  
  
 [[-0.0694, -0.0077, 0.2390, -0.3550],  
 [ 0.2719, 0.3785, 0.0574, 0.1827],  
 [-0.4071, 0.4483, 0.1970, 0.1336],  
 [-0.3864, 0.1077, 0.0382, -0.0352]],  
  
 [[ 0.0638, 0.1997, 0.2277, -0.0291],  
 [-0.0980, 0.2574, 0.0227, -0.0956],  
 [ 0.0505, -0.0889, 0.1744, -0.3381],  
 [ 0.2274, 0.1754, -0.1307, -0.0301]]]], grad\_fn=<SliceBackward0>)

#### 4）训练生成器

在训练判别器计算损失函数时，是最大化训练数据集真图像为真，生成假图像为假的概率。训练生成器时，损失函数计算的方式是生成假图像为真的概率，即 BCELoss 输入参数taget的值为真标签值1，对应代码为tst(im\_x\_G\_D\_sigmoid,label\_real)。然后执行反向传播，更新网络。

generator.zero\_grad()  
im\_x\_G\_D4G=critic(im\_x\_G)  
im\_x\_G\_D4G\_sigmoid=torch.sigmoid(im\_x\_G\_D4G)  
  
im\_x\_G\_D4G\_sigmoid\_tst\_real=tst(im\_x\_G\_D4G\_sigmoid,label\_real)  
print(im\_x\_G\_D4G\_sigmoid\_tst\_real)

TensorBase(0., grad\_fn=<AliasBackward0>)

im\_x\_G\_D4G\_sigmoid\_tst\_real.backward()  
opt\_G=Adam(generator.parameters(), lr=lr)  
opt\_G.step()

#### 5) 训练模型（WGAN方式）

Goodfellow,I.J.等人提出 GAN 时给出了伪代码[1]如下：

**Algorithm 1**Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator,  *k*, is a hyperparameter. We used *k*=1, the least expensive option, in our experiments.

**for**number of training iterations**do**

**for***k* steps**do**

• Sample minibatch of *m* noise samples {***z***(1),…,***z***(*m*)} from noise prior*pg*​(***z***).

• Sample minibatch of *m* examples{***x***(1),…,***x***(*m*)} from data generating distribution *p*data ​(***x***).

• Update the discriminator by ascending its stochastic gradient: ∇*θd*​​*m*1​∑*i*=1*m*​[log*D*(***x***(*i*))+log(1−*D*(*G*(***z***(*i*))))].

**end for**

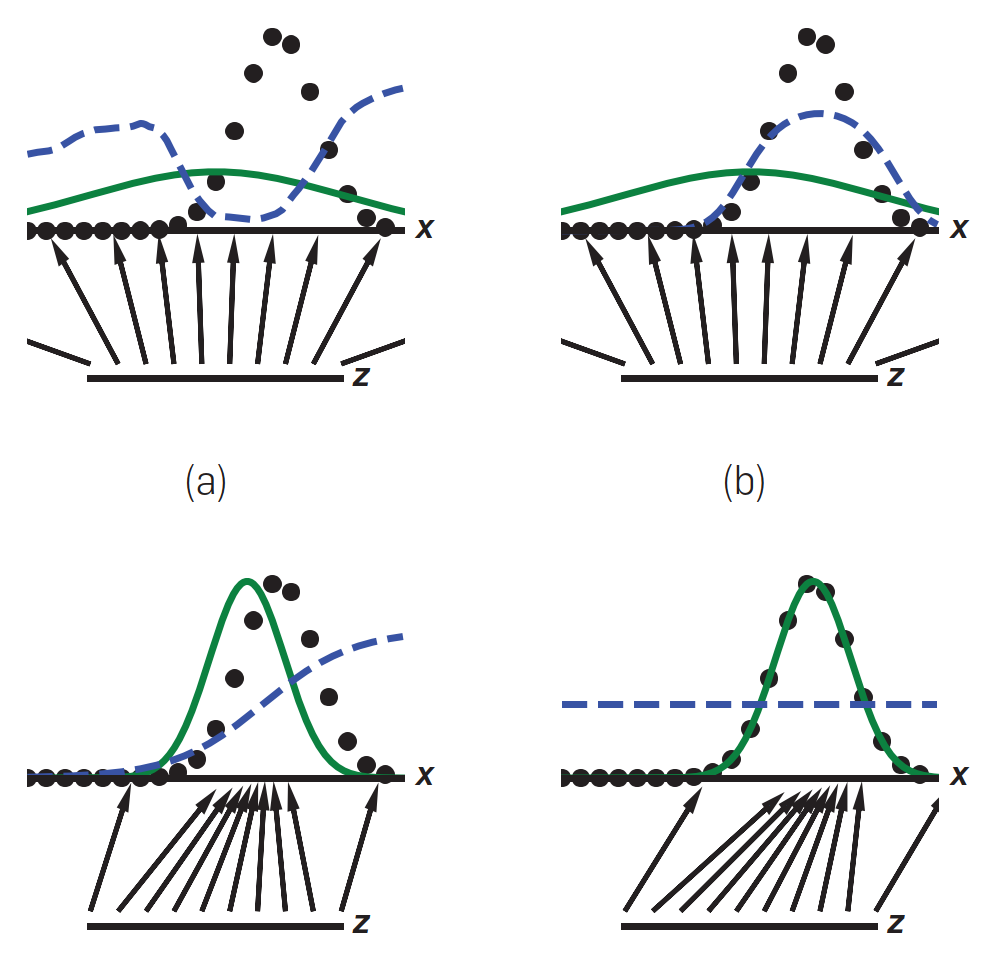
• Sample minibatch of *m* noise samples {***z***(1),…,***z***(*m*)} from noise prior *pg*​(***z***).

• Update the generator by descending its stochastic gradient: ∇*θg*​​*m*1​∑*i*=1*m*​log(1−*D*(*G*(***z***(*i*)))).

**end for**

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

并用图示拟合一维高斯分布的方法解释了 GAN 的计算过程[14]，



将的目标理解为学习真实分布的逆累积分布函数（cumulative distribution function，CDF）的缩放（scaling）过程。GAN 通过同时更新（蓝色虚线）进行训练，以便区分真实分布（黑色虚线）样本和生成分布（绿色实线）样本。下面的水平线是对进行采样的域（domain），本例为均匀采样。上面的水平线是的一部分。向上的箭头显示映射是如何将非均匀分布强加到变换后的样本上。在的高密度区域收缩，并扩展到低密度区域。

（a）这个案例中，被初始化为标准正态分布（unit Gaussian）；为随机初始化的深度神经网络定义；

（b）假设保持不变，训练收敛至。（实际两者同时训练）；

（c）假设和逐渐训练了一段时间，生成的样本沿着增加的方向流动，以便达到可能被分类为真实分布的区域。同时，对应的参数更新，更新的参数；

（d）在纳什均衡（Nash equilibrium）中，由于，两个参与者都不能提高收益，无法区分这两个分布，即这个常量函数（constant function）表明所有点都同样可能来自任意分布。

上述原始 GAN 达到纳什均衡的理想结果通常很困难，损失函数的误差曲线降不下去（梯度消失，例如训练的越好，的梯度消失越严重）或者波动较大（梯度不稳定），很难从生成器和判别器的损失函数误差曲线上判断训练好坏程度。[15]。Arjovsky, M.[9]等提出了 WGAN， 应用 wasserstein（或称为 Earth-Mover (EM)） 距离解决了上述问题。

* EM 距离（wasserstein-1 distance）

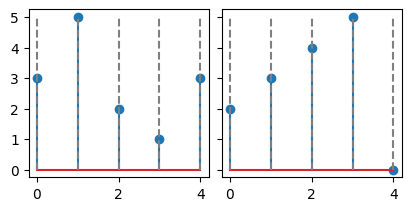
EM 距离是用来衡量两个概率分布之间差异的方法，为由一个分布转化为另一个分布的最小成本。SciPy库的scipy.stats.wasserstein\_distance(u\_values, v\_values, u\_weights=None, v\_weights=None)方法提供了 EM 距离计算，其中，u\_values和v\_values为（经验）分布中观测到的值；u\_weights和v\_weights可以为每个值指定权重。EM 距离计算公式按 Arjovsky, M.论文中的表述给出为：，式中，为边缘分布（marginals）分别为和的所有联合分布（joint distributions）γ (x,y) γ(x,y) 表示必须将多少“质量（mass）”从传输到，以便将分布转化为分布。

关于计算两个概率分布之间差异（距离）的方法可以同时参考*标记距离*一章距离度量部分。

下面应用wasserstein\_distance方法，假设分布和，计算两种情况下的 EM 距离。将横轴值作为观测到的值，对应变量vals；将纵轴值作为对应观测值观测到的频数，即和分布。

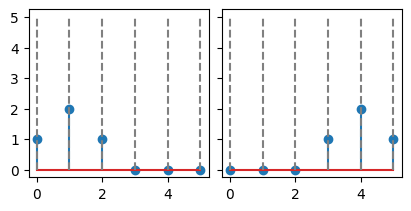
from scipy.stats import wasserstein\_distance  
import matplotlib.pyplot as plt  
  
P=[3,5,2,1,3]  
Q=[2,3,4,5,0]  
vals=[0,1,2,3,4]  
print(wasserstein\_distance(vals,vals,P,Q))  
  
fig,axs=plt.subplots(1,2,layout='constrained',sharey=True,figsize=(4,2))  
axs[0].stem(vals, P)  
axs[1].stem(vals,Q)  
  
axs[0].vlines(x=vals,ymin=0,ymax=5,colors='gray',linestyles='dashed')  
axs[1].vlines(x=vals,ymin=0,ymax=5,colors='gray',linestyles='dashed')  
plt.show()

0.5714285714285714



P=[1,2,1]  
Q=[1,2,1]  
vals\_P=[0,1,2]  
vals\_Q=[3,4,5]  
vals=vals\_P+vals\_Q  
print(wasserstein\_distance(vals\_P,vals\_Q,P,Q))  
  
\_,axs=plt.subplots(1,2,layout='constrained',sharey=True,figsize=(4,2))  
axs[0].stem(vals,P+[0,0,0])  
axs[1].stem(vals,[0,0,0]+Q)  
  
axs[0].vlines(x=vals,ymin=0,ymax=5,colors='gray',linestyles='dashed')  
axs[1].vlines(x=vals,ymin=0,ymax=5,colors='gray',linestyles='dashed')  
plt.show()

3.0



从上述第2个假设的分布可以得知，即使和两个分布没有重叠，EM 距离计算结果也能够反映二者的远近，因此用梯度下降法优化 EM 距离，表示为，可以提供有意义的梯度。结合到 WGAN，的损失函数定义为：；的损失函数定义为：，式中，为。的损失函数可以指示训练进程，其值越小，表示生成分布到真实分布的 EM 距离越小，GAN 模型训练的越好。

WGAN 的伪代码[9]如下：

**Algorithm 2**WGAN, our proposed algorithm. All experiments in the paper used the default values *α*=0.00005,*c*=0.01,*m*=64,*n*critic ​=5

**Require:***α*, the learning rate.  *c*, the clipping parameter.  *m*, the batch size.  critic *n*critic ​, the number of iterations of the critic per generator iteration.

**Require:** *w*0​, initial critic parameters. *θ*0​, initial generator's parameters.

**while***θ* has not converged**do**

**for** *t*=0,…,*n*critic ​**do**

Sample {*x*(*i*)}*i*=1*m*​∼P*r*​ a batch from the real data.

Sample {*z*(*i*)}*i*=1*m*​∼*p*(*z*) a batch of prior samples.

*gw*​←∇*w*​[*m*1​∑*i*=1*m*​*fw*​(*x*(*i*))−*m*1​∑*i*=1*m*​*fw*​(*gθ*​(*z*(*i*)))]

*w*←*w*+*α*⋅RMSProp(*w*,*gw*​)

*w*←clip(*w*,−*c*,*c*)

**end for**

Sample {*z*(*i*)}*i*=1*m*​∼*p*(*z*) a batch of prior samples.

*gθ* ​←−∇*θ*​*m*1​∑*i*=1*m*​*fw*​(*gθ*​(*z*(*i*)))

*θ*←*θ*−*α*⋅RMSProp(*θ*,*gθ*​)

**end while**

因为遥感（航拍）地物繁多，形式复杂且模式近似，用于 MNIST、CIFAR10 等自然图像及 Imagenet 数据集表现良好的 GAN 模型，可能对高分辨率的遥感影像分布学习生成失去作用[16]。上述阐释的 WGAN 表现也同样并不理想。

在训练过程在中，增加了SaveModelCallback回调函数，用于根据指定的损失函数monitor，变化幅度min\_delta，保存的文件名fname，根据训练情况保存模型参数；同时也使用leaner.save()的方法保存模型，用leaner.load()可以加载保存的模型。观察下述训练过程的crit\_loss变化，当学习率配置为lr=1e-5时，crit\_loss能够稳步下降。

深度学习模型的训练在[CoLab](https://colab.research.google.com/)中完成。

cbs=[SaveModelCallback(monitor='crit\_loss',min\_delta=0.01, with\_opt=True,fname='naip\_wgan\_b\_ckpt',)] # EarlyStoppingCallback(monitor='gen\_loss', min\_delta=0.01, patience=10),  
learn=GANLearner.wgan(dls, generator, critic, opt\_func=RMSProp,  
 path=ckpts\_path,  
 cbs=cbs,  
 lr=1e-5, # 2e-4  
 clip=0.01,  
 switcher=FixedGANSwitcher(n\_crit=5, n\_gen=1),  
 switch\_eval=False  
 )  
  
learn.recorder.train\_metrics=True  
learn.recorder.valid\_metrics=False

迭代5轮次，从显示结果可以看到色彩、模式、对象形态还都没有显现。

learn.fit\_one\_cycle(5,wd=0.)

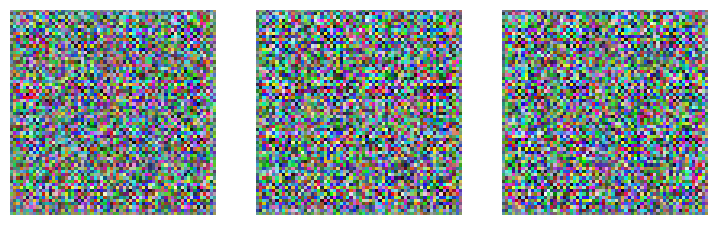
| **epoch** | **train\_loss** | **gen\_loss** | **crit\_loss** | **time** |
| --- | --- | --- | --- | --- |
| 0 | -0.027092 | 0.016904 | -0.022706 | 00:26 |
| 1 | -0.268401 | 0.159419 | -0.267501 | 00:17 |
| 2 | -0.499596 | 0.321357 | -0.620705 | 00:17 |
| 3 | -0.591036 | 0.385114 | -0.764294 | 00:17 |
| 4 | -0.613021 | 0.403253 | -0.803440 | 00:18 |

Better model found at epoch 0 with crit\_loss value: -0.02270558290183544.  
Better model found at epoch 1 with crit\_loss value: -0.2675013840198517.  
Better model found at epoch 2 with crit\_loss value: -0.6207051873207092.  
Better model found at epoch 3 with crit\_loss value: -0.764293909072876.  
Better model found at epoch 4 with crit\_loss value: -0.8034396767616272.

-1.3153190612792969.

0.0058530885726213455.

learn.show\_results(max\_n=3, ds\_idx=0)



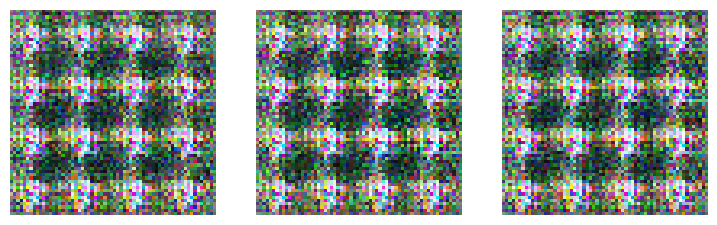
继续迭代10轮次，色彩开始趋向于真实样本，对象形态雏形隐约显现。

learn.fit\_one\_cycle(10)

| **epoch** | **train\_loss** | **gen\_loss** | **crit\_loss** | **time** |
| --- | --- | --- | --- | --- |
| 0 | -0.606376 | 0.399832 | -0.815916 | 00:17 |
| 1 | -0.689078 | 0.439710 | -0.882295 | 00:17 |
| 2 | -0.789199 | 0.498843 | -1.008572 | 00:17 |
| 3 | -0.875738 | 0.554583 | -1.128081 | 00:18 |
| 4 | -0.930151 | 0.591104 | -1.206856 | 00:17 |
| 5 | -0.965422 | 0.614313 | -1.255906 | 00:17 |
| 6 | -0.989825 | 0.631252 | -1.290740 | 00:17 |
| 7 | -1.004018 | 0.641229 | -1.311686 | 00:18 |
| 8 | -1.010721 | 0.646002 | -1.321508 | 00:18 |
| 9 | -1.012111 | 0.647487 | -1.324434 | 00:17 |

Better model found at epoch 0 with crit\_loss value: -0.8159159421920776.  
Better model found at epoch 1 with crit\_loss value: -0.8822952508926392.  
Better model found at epoch 2 with crit\_loss value: -1.0085723400115967.  
Better model found at epoch 3 with crit\_loss value: -1.1280813217163086.  
Better model found at epoch 4 with crit\_loss value: -1.206856369972229.  
Better model found at epoch 5 with crit\_loss value: -1.255906105041504.  
Better model found at epoch 6 with crit\_loss value: -1.29073965549469.  
Better model found at epoch 7 with crit\_loss value: -1.311686396598816.  
Better model found at epoch 9 with crit\_loss value: -1.32443368434906.

learn.show\_results(max\_n=3, ds\_idx=0)



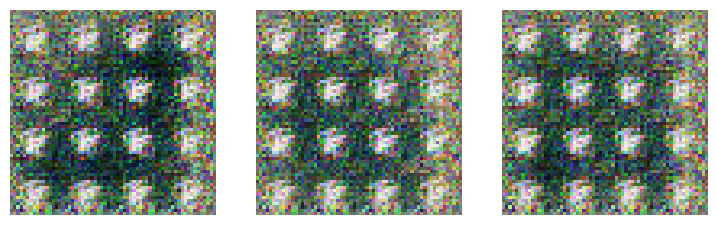
继续迭代10轮次，crit\_loss稳步下降收敛。色彩继续趋于真实分布。

learn.fit\_one\_cycle(10)

| **epoch** | **train\_loss** | **gen\_loss** | **crit\_loss** | **time** |
| --- | --- | --- | --- | --- |
| 0 | -0.982120 | 0.635491 | -1.327267 | 00:17 |
| 1 | -1.029104 | 0.657207 | -1.343396 | 00:18 |
| 2 | -1.053444 | 0.671278 | -1.372857 | 00:17 |
| 3 | -1.074646 | 0.685937 | -1.402718 | 00:17 |
| 4 | -1.095327 | 0.696518 | -1.427755 | 00:17 |
| 5 | -1.111538 | 0.706379 | -1.449256 | 00:18 |
| 6 | -1.126531 | 0.715450 | -1.470485 | 00:17 |
| 7 | -1.136540 | 0.722365 | -1.485029 | 00:17 |
| 8 | -1.141410 | 0.725229 | -1.491761 | 00:17 |
| 9 | -1.142409 | 0.726656 | -1.494095 | 00:17 |

Better model found at epoch 0 with crit\_loss value: -1.3272665739059448.  
Better model found at epoch 1 with crit\_loss value: -1.3433961868286133.  
Better model found at epoch 2 with crit\_loss value: -1.3728567361831665.  
Better model found at epoch 3 with crit\_loss value: -1.402718424797058.  
Better model found at epoch 4 with crit\_loss value: -1.427754521369934.  
Better model found at epoch 5 with crit\_loss value: -1.4492558240890503.  
Better model found at epoch 6 with crit\_loss value: -1.4704854488372803.  
Better model found at epoch 7 with crit\_loss value: -1.4850293397903442.

learn.show\_results(max\_n=3, ds\_idx=0)



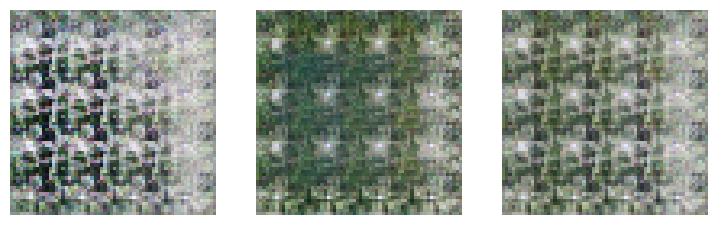
继续迭代200轮次，色彩基本同真实图像，对象形态继续得到优化。

learn.fit\_one\_cycle(200)

| **epoch** | **train\_loss** | **gen\_loss** | **crit\_loss** | **time** |
| --- | --- | --- | --- | --- |
| 0 | -1.103175 | 0.713960 | -1.490784 | 00:17 |
| 1 | -1.140194 | 0.725834 | -1.492246 | 00:18 |
| ... | ... | ... | ... | ... |
| 197 | -1.182713 | 0.755221 | -1.547309 | 00:19 |
| 198 | -1.182745 | 0.755224 | -1.547347 | 00:18 |
| 199 | -1.182684 | 0.755274 | -1.547368 | 00:18 |

Better model found at epoch 155 with crit\_loss value: -1.546173334121704.

learn.show\_results(max\_n=3, ds\_idx=0)



当训练时间较长时，可以保存模型，避免训练参数丢失。当再需要继续训练时，可以加载模型。

learn.save(ckpts\_path/'naip\_wgan\_b\_learn',with\_opt=True)

Path('/content/gdrive/MyDrive/Colab\_Notebooks/model\_checkpoint/ckpts4NaipGAN/naip\_wgan\_b\_learn.pth')

learn.load(ckpts\_path/'naip\_wgan\_b\_learn',with\_opt=True)

<fastai.vision.gan.GANLearner at 0x7ff1d17cd130>

加载保存的训练模型后，继续训练。

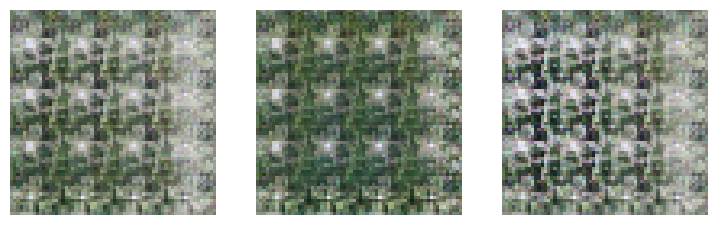
learn.recorder.train\_metrics=True  
learn.recorder.valid\_metrics=False  
  
learn.fit\_one\_cycle(100)

| **epoch** | **train\_loss** | **gen\_loss** | **crit\_loss** | **time** |
| --- | --- | --- | --- | --- |
| 0 | -1.143211 | 0.743186 | -1.546363 | 00:18 |
| 1 | -1.180329 | 0.754740 | -1.546314 | 00:18 |
| ... | ... | ... | ... | ... |
| 98 | -1.157531 | 0.740918 | -1.515097 | 00:17 |
| 99 | -1.157991 | 0.741140 | -1.515386 | 00:18 |

Better model found at epoch 0 with crit\_loss value: -1.546363115310669.

迭代 325 轮次后，与截止至 225 时显示结果近似，学习效果已不明显。

learn.show\_results(max\_n=3, ds\_idx=0)



保存最后训练的模型（为新文件名，未覆盖之前保存的文件）。

learn.save(ckpts\_path/'naip\_wgan\_c\_learn',with\_opt=True)

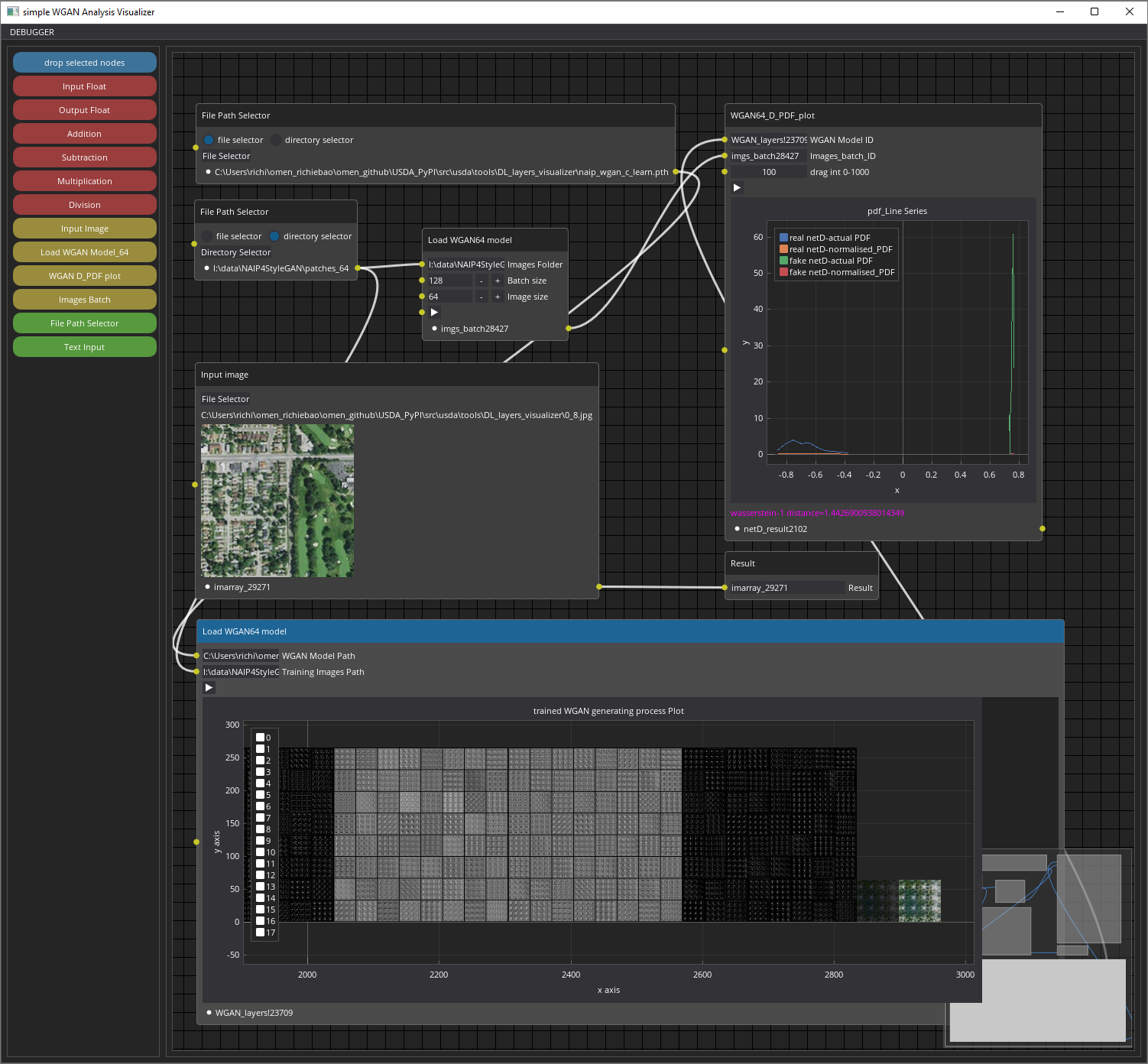
Path('/content/gdrive/MyDrive/Colab\_Notebooks/model\_checkpoint/ckpts4NaipGAN/naip\_wgan\_c\_learn.pth')

## 3.4.3 计算分析工具的建构——Dear PyGUI

建立计算分析工具的主要目的有，集成一些经常用到、重复用到或者专门用到的计算代码，方便以 GUI（Graphical user interface） 可视化的图形用户界面方式调用；将研究内容以 GUI 工具的方式演示，具有可操作性、交互性、便捷性等特点；为未掌握编程语言的团队成员或相关使用者提供计算分析工具软件。

基于 Python 编程语言的 GUI 建构工具日益丰富，例如在*基于NDVI指数解译影像与建立采样工具*一章使用[Tkinter](https://docs.python.org/3/library/tkinter.html)⑯建立遥感影像的采样工具；结合[Gradio](https://gradio.app/)⑰演示机器学习模型；用[PyQt](https://wiki.python.org/moin/PyQt)⑱建构软件；[Pygame](https://www.pygame.org/news)⑲设计游戏；及[MD Python Designer](https://labdeck.com/python-designer/)⑳、[PySimpleGUI](https://www.pysimplegui.org/en/latest/)㉑、[pyglet](https://pyglet.org/)㉒、[PySide](https://www.pythonguis.com/pyside6/)㉓等。[Dear GUI（DPG）](https://dearpygui.readthedocs.io/en/latest/index.html)㉔则更倾向于为数据分析人员使用，是一个易于使用、动态、GPU 加速、跨平台的 Python 图形用户界面工具包，功能包括传统的 GUI 元素，例如按钮、单选按钮、菜单和创建功能布局的各种方法；含有种类繁多的动态图、表格、绘图、调试器和多个资源查看器；同时包含节点（Node）编辑器，建构可视化节点编程工具；且其框架弹性自由，表达形式布局巧妙，形式新颖。DPG 非常适用于创建简单用户界面及开发复杂且要求苛刻的图形界面，为开发科学、工程、游戏、数据科学和其他需要快速交互界面的应用程序提供坚实的框架。因此，对于数据分析工作，在平衡快速、方便的建构，代码书写的Python化、和向专门软件的转化，及 GUI 架构的自由性，使用 DPG 建立相关研究的计算分析工具。

为了方便查看训练后的 WGAN 模型中网络各层数据演化情况，用 DGU 建立simple WGAN Analysis Visualizer计算分析工具，构建Load WGAN64 model节点；比较网络一批真实图像数据计算结果的分布情况和生成同等数量一批生成假数据计算结果分布情况，并计算EM 距离，构建WGAN64\_D\_PDF\_plot节点；同时构建有图像输入显示节点Input Image，数据集一批建构Image Batch，文件选择节点File Path Selector等辅助节点。DPG 的节点编辑器基础架构选择了[DearPyGUI NodeEditor Template](https://codeberg.org/LuminousLizard/DearPyGUI_NodeEditor_Template)㉕提供的代码。



上述 WGAN 的 网络包括17个单层，每层生成新数据的形状可能不同，由输入的 100 维（Channel，通道数）(1,1)大小的噪声数据（latent space）降维到 3 维（RGB）(64,64)大小的（图像）数据。通道数大于3维度时，以多个单通道的(size,size)大小网格状显示（最多100个），例如对于[1, 128, 16, 16]形状数据(bs,ch,size,size)，有1个样本，128个通道（维度），大小为，则取前 100 维，每维显示的灰度图像，并排列为网格显示这100维数据。对于最后的[1,3,64,64]则按RGB方式显示为彩色图像。

基于 网络的数据分布和EM 距离计算，输入的为真实图像数据集，而生成图像根据输入真实图像数据的数量构建噪声数据，并由网络生成，用于网络的输入。从打印的真实图像和生成图像的分布曲线可以判断，当前训练的 WGAN 模型的生成图像并不理想。

simple WGAN Analysis Visualizer计算分析工具置于USDA下的tools\DL\_layers\_visualizer内。执行DL\_layers\_visualizer.py模块运行。

注释（Notes）：

① Chicago Data Portal，（<https://data.cityofchicago.org/Buildings/Building-Footprints-current-/hz9b-7nh8>）。

② Chicago Metropolitan Agency for Planning, CMAP，（<https://www.cmap.illinois.gov/data/land-use/inventory>）。

③ ArcGIS-Chicago Regional Land Cover，（<https://www.arcgis.com/home/item.html?id=782adcff882d4f09a227b509dcaa1628>）。

④ USGS EarthExploer，（<https://earthexplorer.usgs.gov/>）。

⑤ Earth Observation Group，（<https://eogdata.mines.edu/products/vnl/>）。

⑥ Pix2pix，（<https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix>）。

⑦ NAIP: National Agriculture Imagery Program，（<https://planetarycomputer.microsoft.com/dataset/naip#Example-Notebook>）。

⑧ StyleGAN3，（<https://github.com/NVlabs/stylegan3>）。

⑨ MarsGAN，（<https://github.com/kheyer/MarsGAN/blob/master/StyleGAN.ipynb>）。

⑩ PyTorch，（<https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html>）。

⑪ fastai，（<https://docs.fast.ai/>）。

⑫ DCGAN TUTORIAL，（<https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html>）。

⑬ fastai-GAN，（<https://docs.fast.ai/vision.gan.html>）。

⑭ PyTorch Lightning，（<https://lightning.ai/docs/pytorch/stable/>）。

⑮ TorchGeo，（<https://pytorch.org/blog/geospatial-deep-learning-with-torchgeo/>）。

⑯ Tkinter，（<https://docs.python.org/3/library/tkinter.html>）。

⑰ Gradio，（<https://gradio.app/>）。

⑱ PyQt，（<https://wiki.python.org/moin/PyQt>）。

⑲ Pygame，（<https://www.pygame.org/news>）。

⑳ MD Python Designer，（<https://labdeck.com/python-designer/>）。

㉑ PySimpleGUI，（https://www.pysimplegui.org/en/latest/）。

㉒ pyglet，（https://pyglet.org/）。

㉓ PySide，（https://www.pythonguis.com/pyside6/）。

㉔ Dear GUI（DPG），（https://dearpygui.readthedocs.io/en/latest/index.html）。

㉕ DearPyGUI NodeEditor Template，（https://codeberg.org/LuminousLizard/DearPyGUI\_NodeEditor\_Template）。

参考文献（References）:

[1] Goodfellow, I. J. et al. Generative Adversarial Networks. Preprint at (2014).

[2] gans-awesome-applications，<https://github.com/nashory/gans-awesome-applications>.

[3] Imdat As, Prithwish Basu, Pratap Talwar. Artificial Intelligence in Urban Planning and Design.Imprint: Elsevier (Imprint: Elsevier).

[4] Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T. & Efros, A. A. Context Encoders: Feature Learning by Inpainting. in Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition vols 2016-December 2536–2544 (IEEE Computer Society, 2016).

[5] Image super-resolution through deep learning, <https://github.com/david-gpu/srez>.

[6] 包瑞清.基于机器学习的风景园林智能化分析应用研究[J].风景园林,2019,26(05):29-34.DOI:10.14085/j.fjyl.2019.05.0029.06.

[7] Karras, T. et al. Alias-Free Generative Adversarial Networks. https://nvlabs.github.io/stylegan3.

[8] Radford, A., Metz, L. & Chintala, S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. (2015).

[9] Arjovsky, M., Chintala, S. & Bottou, L. Wasserstein GAN. (2017).

[10] Radford, A., Metz, L. & Chintala, S. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. (2015).

[11] Zeiler, M. D., Krishnan, D., Taylor, G. W. & Fergus, R. Deconvolutional Networks.

[12] Dumoulin, V. & Visin, F. A guide to convolution arithmetic for deep learning. (2016).

[13] Ioffe, S. & Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. (2015).

[14] Goodfellow, I. et al. Generative adversarial networks. Commun ACM 63, 139–144 (2020).

[15] Arjovsky, M. & Bottou, L. Towards Principled Methods for Training Generative Adversarial Networks. (2017).

[16] Sun, H. et al. HRPGAN: A GAN-based Model to Generate High-resolution Remote Sensing Images. in IOP Conference Series: Earth and Environmental Science vol. 428 (Institute of Physics Publishing, 2020).