```
# -*- coding: utf-8 -*-
"""CnovLSTM.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1FXyL49PU3Fkz5M86wlP654fMFOlM5Bqt
"""

# Commented out IPython magic to ensure Python compatibility.
#@title mount
from google.colab import drive
drive.mount('/content/drive')
# %cd "/content/drive/MyDrive/Colab Notebooks/植被覆盖率预测"

#@title selece model and map
import ipywidgets as widgets
from IPython.display import display

# 定义全局变量
model_type = None
map_type = None

# 创建下拉菜单
model_dropdown = widgets.Dropdown(
    options=['CNN', 'LSTM', 'CNNLSTM', 'Attention', 'ViT_Trans', 'ViT_LSTM'],
    description='Model:'
)

map_dropdown = widgets.Dropdown(
    options=['FVC', 'LULC', 'RSEI'],
    description='Map Type:'
)

# 创建按钮
confirm_button = widgets.Button(description='Confirm')

# 显示控件
display(model_dropdown, map_dropdown, confirm_button)

# 定义训练函数
def train_model(model, map_t):
    global model_type, map_type  # 使用 global 关键字声明全局变量
    model_type = model
    map_type = map_t

# 绑定按钮点击事件
confirm_button.on_click(lambda b: train_model(model_dropdown.value, map_dropdown.
value))

import shutil
for model_type in ['CNN', 'LSTM', 'CNNLSTM', 'Attention']:
  base_dir = f'outputs/plots/{model_type}'
  for i in os.listdir(base_dir):
    if i != "Report":
```

```
      continue
    print(os.path.join(base_dir, i))
    shutil.rmtree(os.path.join(base_dir, i))

print(model_type, map_type)

#@title load data
import torch
from torch.utils.data import Dataset, DataLoader
import numpy as np
from tqdm.notebook import tqdm
import random
from torch.utils.data import DataLoader, WeightedRandomSampler

class MyDataSet(Dataset):
    def __init__(self, data, mask, seed, window_size=15, region_size=10,↙
sample_size=100000, split_ratios=(0.8, 0.1, 0.1), subset='train'):
        """
        自定义数据集类，用于动态生成数据，排除所有标签值为0的点。

        参数：
        - data: 输入的三维数据 (年数, 高, 宽)，形状为 (20, 4416, 5786)
        - mask: 二维蒙版 (4416, 5786)，值为0表示对应位置固定为零，不用于训练
        - window_size: 用于预测的历史年份数，默认为6
        - region_size: 输入区域大小，默认为10
        """
        self.data = data
        self.mask = mask
        self.seed = seed
        self.subset = subset
        self.sample_size = sample_size
        self.split_ratios = split_ratios
        self.window_size = window_size
        self.region_size = region_size
        self.height, self.width = data.shape[1], data.shape[2]
        self.year_range = data.shape[0] - window_size  # 可用的年份范围
        self.offset = region_size // 2
        self.sampler = None

        # 根据mask找到所有有效的 (row, col) 坐标
        self.valid_spatial_indices = self._get_valid_spatial_indices()

        self.class_dict = {}
        if subset == 'train':
          self.sampler = self.compute_class_weights()

        # 按 split_ratios 划分训练集、验证集和测试集

    def _get_valid_spatial_indices(self):
        """
        根据mask筛选出所有可能的非零标签点的 (row, col) 坐标
        打乱顺序，再顺序切片出需要的样本数
        再对应的训练集、验证集和测试集进行划分
        """
        # indices = []
```

```
        # for i in tqdm(range(self.offset, self.height - self.offset)):
        #     for j in range(self.offset, self.width - self.offset):
        #         # 如果mask为1，表示该位置可以有非零标签
        #         if self.mask[i, j] != 0:
        #             indices.append((i, j))

        # 提取非零坐标数组
        indices = np.nonzero(self.mask[self.offset:self.height - self.offset, self.↙
offset:self.width - self.offset])
        indices = list(zip(indices[0] + self.offset, indices[1] + self.offset))

        # print(self.data.shape, len(indices))
        # indices_array = np.array(indices)
        # extracted_data = self.data[:, indices_array[:, 0], indices_array[:, 1]]
        # print(extracted_data.shape)

        random.seed(self.seed)
        random.shuffle(indices)
        print(f"Valid spatial indices({self.subset}) done")
        print('\tAll indices:', len(indices))
        indices = indices[:self.sample_size]
        print(f'\tSelected indices:', len(indices))

        train_split = int(self.split_ratios[0] * len(indices))
        val_split = int(self.split_ratios[1] * len(indices)) + train_split

        if self.subset == 'train':
          train_indices = indices[:train_split]
          print("\tTraining samples:", len(train_indices))
          return train_indices
        elif self.subset == 'val':
          val_indices = indices[train_split:val_split]
          print("\tValidation samples:", len(val_indices))
          return val_indices
        elif self.subset == 'test':
          test_indices = indices[val_split:]
          print("\tTesting samples:", len(test_indices))
          return test_indices

    def compute_class_weights(self):
        """
        统计训练集的类别数量并计算类别权重。
        """
        for year_idx in tqdm(range(self.window_size, self.year_range + self.↙
window_size), desc='Calculating weight...'):
            for row, col in self.valid_spatial_indices:
                label = self.data[year_idx, row, col] - 1  # 获取标签
                if label in self.class_dict:
                    self.class_dict[label] += 1
                else:
                    self.class_dict[label] = 1

        # 计算权重：总样本数除以每个类别的数量
        total_count = sum(self.class_dict.values())
        self.class_weights = {label: total_count / count for label, count in self.↙
```

```
class_dict.items()}

        print("Class counts:", self.class_dict)
        print("Class weights:", self.class_weights)
        sample_weights = []
        for year_idx in tqdm(range(self.window_size, self.year_range + self.↙
window_size), desc='Assigning weight...'):
            for row, col in self.valid_spatial_indices:
                label = self.data[year_idx, row, col] - 1  # 获取标签
                sample_weights.append(self.class_weights[label])

        # 创建加权随机采样器
        sampler = WeightedRandomSampler(
            weights=sample_weights,
            num_samples=len(sample_weights),
            replacement=True
        )
        return sampler

    def get_classes(self):
        return self.class_dict

    def __len__(self):
        # 数据集的长度是年份数乘以有效的空间位置数
        return self.year_range * len(self.valid_spatial_indices)

    def __getitem__(self, idx):
        # 根据数据集索引确定 year 和 (row, col) 的位置
        spatial_idx = idx % len(self.valid_spatial_indices)
        year_idx = idx // len(self.valid_spatial_indices) + self.window_size
        row, col = self.valid_spatial_indices[spatial_idx]

        # 根据年份和坐标获取实际输入数据 (window_size, region_size, region_size)
        input_data = self.data[year_idx - self.window_size:year_idx,
                               row - self.offset:row + self.offset + 1,
                               col - self.offset:col + self.offset + 1]

        # 获取标签 (目标年份的中心点值)
        label_data = self.data[year_idx, row, col]-1
        if label_data in self.class_dict.keys():
          self.class_dict[label_data] += 1
        else:
          self.class_dict[label_data] = 1

        # 转换为Tensor
        input_tensor = torch.tensor(input_data, dtype=torch.float32)
        label_tensor = torch.tensor(label_data, dtype=torch.long)
        return input_tensor, label_tensor

# 读取 npz 文件并加载数据
fvc_data = np.load(f'{map_type}.npz')['arr_0']
# fvc_data = data['arr_0']
mask = np.load('whole_mask.npz')['arr_0']
# print(fvc_data.shape)  # 确认数据形状为 (20, 4416, 5786)
```

```
print("Loading data done")

# 创建Dataset实例并使用DataLoader按批次加载数据
seed = random.randint(0, 100000)
sample_size = 200000
train_dataset = MyDataSet(fvc_data, mask, seed, subset='train',↙
sample_size=sample_size)
val_dataset = MyDataSet(fvc_data, mask, seed, subset='val', sample_size=sample_size)
test_dataset = MyDataSet(fvc_data, mask, seed, subset='test', sample_size=sample_size)

sampler = train_dataset.sampler

# for i in tqdm(train_dataset):
#     continue
#     # if int(i[1]) == 0:
#       # continue
#     # print(i[0].shape, i[1])

# print(seed)
# print(train_dataset.show_class())

"""# Models"""

#@title CNN
import torch
import torch.nn as nn

class PureCNNModel(nn.Module):
    def __init__(self, map_type):
        super(PureCNNModel, self).__init__()

        # 卷积层
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=0, stride=2)  # 输入1通道，↙
输出32通道
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=0, stride=2)  # 输入32通↙
道，输出64通道

        self.bn1 = nn.BatchNorm2d(32)
        self.bn2 = nn.BatchNorm2d(64)

        # Dropout层
        self.dropout_conv = nn.Dropout2d(0.5)  # 卷积层后的Dropout

        # 计算展平后的维度
        self.flat_dim = 64 * 2 * 2  # 假设输入大小为11x11，经过卷积后展平

        # 全连接层
        if map_type == "LULC":
            self.fc = nn.Linear(self.flat_dim, 8)  # 映射到8个分类
        else:
            self.fc = nn.Linear(self.flat_dim, 6)  # 映射到6个分类

    def forward(self, x):
        batch_size, time_steps, width, height = x.shape  # x.shape = [batch_size,↙
time_steps, 11, 11]
```

```
        # 合并时间步到批次维度
        x = x.view(batch_size * time_steps, 1, width, height)  # [batch_size *↙
time_steps, 1, 11, 11]

        # CNN提取特征
        x = torch.relu(self.bn1(self.conv1(x)))
        x = torch.relu(self.bn2(self.conv2(x)))
        x = self.dropout_conv(x)

        # 展平
        x = x.view(batch_size * time_steps, -1)  # 展平 [batch_size * time_steps,↙
flat_dim]

        # 全连接层
        x = self.fc(x)  # 映射到分类 [batch_size * time_steps, num_classes]

        # 恢复批次和时间步的分离
        x = x.view(batch_size, time_steps, -1)  # [batch_size, time_steps,↙
num_classes]

        # 平均时间步的分类结果
        x = x.mean(dim=1)  # [batch_size, num_classes]

        return x

# model = PureCNNModel(map_type=map_type)
# input_tensor = torch.randn(32, 15, 11, 11)  # 假设批次大小为32，时间步为15
# output_tensor = model(input_tensor)
# print("Output shape:", output_tensor.shape)  # 输出应为 [32, num_classes]

#@title LSTM
import torch
import torch.nn as nn

class PureLSTMModel(nn.Module):
    def __init__(self, map_type, input_dim=11 * 11, hidden_size=128, num_layers=2,↙
num_classes=6):
        super(PureLSTMModel, self).__init__()

        # LSTM 层
        self.lstm = nn.LSTM(
            input_size=input_dim,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True
        )

        # 分类层

        if map_type == "LULC":
            self.fc = nn.Linear(hidden_size, 8)  # 映射到8个分类
        else:
            self.fc = nn.Linear(hidden_size, 6)  # 映射到6个分类
```

```
    def forward(self, x):
        batch_size, time_steps, width, height = x.shape  # x.shape = [batch_size,↙
time_steps, 11, 11]

        # 将每个时间步展平成一维特征
        x = x.view(batch_size, time_steps, -1)  # shape = [batch_size, time_steps, 11↙
* 11]

        # 使用 LSTM 提取序列特征
        lstm_out, _ = self.lstm(x)  # lstm_out shape = [batch_size, time_steps,↙
hidden_size]

        # 取 LSTM 的最后一个时间步的输出
        x = lstm_out[:, -1, :]  # shape = [batch_size, hidden_size]

        # 分类
        x = self.fc(x)  # shape = [batch_size, num_classes]

        return x

# # 测试模型结构
# model = PureLSTMModel(map_type=map_type)
# input_tensor = torch.randn(32, 15, 11, 11)  # 假设批量大小为32
# output_tensor = model(input_tensor)
# print("Output shape:", output_tensor.shape)  # 输出应为 [32, 6]

#@title CNNLSTM
import torch
import torch.nn as nn

class CNN_LSTM_Model(nn.Module):
    def __init__(self, map_type):
        super(CNN_LSTM_Model, self).__init__()
        # 卷積層
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=0, stride=2)  # 每個時間步↙
輸入1通道，輸出32通道
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=0, stride=2)  # 輸入32通↙
道，輸出64通道

        self.bn1 = nn.BatchNorm2d(32)
        self.bn2 = nn.BatchNorm2d(64)

        # Dropout層
        self.dropout_conv = nn.Dropout2d(0.5)  # 卷積層之後的Dropout

        # 計算展平後的維度: 128 * 11 * 11
        # self.flat_dim = 64 * 11 * 11
        self.flat_dim = 64 * 2 * 2

        # LSTM層，用於處理序列數據
        self.lstm = nn.LSTM(input_size=self.flat_dim, hidden_size=128, num_layers=1,↙
batch_first=True)

        # 全連接層
        if map_type == "LULC":
```

```
            self.fc = nn.Linear(128, 8)  # 將LSTM的最後一個時間步的輸出映射到8個分類
        else:
            self.fc = nn.Linear(128, 6)  # 將LSTM的最後一個時間步的輸出映射到6個分類

    def forward(self, x):
        batch_size, time_steps, width, height = x.shape  # x.shape = [8192, 15, 11, ↙
11]

        # 將時間步合併到批次維度，進行批量卷積操作
        x = x.view(batch_size * time_steps, 1, width, height)  # shape = [batch_size *↙
time_steps, 1, 11, 11]

        # CNN提取空間特徵
        x = torch.relu(self.conv1(x))
        # x = self.bn1(x)
        x = torch.relu(self.conv2(x))
        # x = self.bn2(x)
        x = self.dropout_conv(x)
        # print(x.shape)

        # 展平
        x = x.view(batch_size, time_steps, -1)  # shape = [batch_size, time_steps,↙
flat_dim]
        # print(x.shape)

        # 使用LSTM處理序列數據
        lstm_out, _ = self.lstm(x)  # lstm_out shape = [batch_size, time_steps, 128]
        # print(lstm_out.shape)

        # 取LSTM的最後一個時間步的輸出
        x = lstm_out[:, -1, :]  # shape = [batch_size, 128]

        # 全連接層進行分類
        x = self.fc(x)  # shape = [batch_size, 6]

        return x

# # 测试模型结构
# model = CNN_LSTM_Model(map_type=map_type)
# # input_tensor = torch.randn(32, 15, 11, 11)  # 假设批量大小为32
# # output_tensor = model(input_tensor)
# # print("Output shape:", output_tensor.shape)  # 输出应为 [32, 6]

#@title ConvLSTM
import torch
import torch.nn as nn
import torch.nn.functional as F

class ConvLSTMCell(nn.Module):
    def __init__(self, input_size, hidden_size, kernel_size):
        super(ConvLSTMCell, self).__init__()

        self.hidden_size = hidden_size
        self.kernel_size = kernel_size
        self.padding = (kernel_size[0] // 2, kernel_size[1] // 2)
```

```
        # 卷积层：用于计算 LSTM 的输入门、遗忘门、输出门和候选状态
        self.conv_i = nn.Conv2d(input_size + hidden_size, hidden_size,↙
kernel_size=kernel_size, padding=self.padding)
        self.conv_f = nn.Conv2d(input_size + hidden_size, hidden_size,↙
kernel_size=kernel_size, padding=self.padding)
        self.conv_o = nn.Conv2d(input_size + hidden_size, hidden_size,↙
kernel_size=kernel_size, padding=self.padding)
        self.conv_g = nn.Conv2d(input_size + hidden_size, hidden_size,↙
kernel_size=kernel_size, padding=self.padding)

    def forward(self, x, h, c):
        # 拼接输入和上一时刻的隐藏状态
        combined = torch.cat([x, h], dim=1)

        # 计算 LSTM 的四个门
        i = torch.sigmoid(self.conv_i(combined))  # 输入门
        f = torch.sigmoid(self.conv_f(combined))  # 遗忘门
        o = torch.sigmoid(self.conv_o(combined))  # 输出门
        g = torch.tanh(self.conv_g(combined))     # 候选状态

        # 更新细胞状态
        c_new = f * c + i * g

        # 计算新的隐藏状态
        h_new = o * torch.tanh(c_new)

        return h_new, c_new


class ConvLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, kernel_size, num_layers, output_size):
        super(ConvLSTM, self).__init__()

        self.num_layers = num_layers
        self.hidden_size = hidden_size

        # 初始化多个 ConvLSTM 层
        self.layers = nn.ModuleList([
            ConvLSTMCell(input_size if i == 0 else hidden_size, hidden_size,↙
kernel_size)
            for i in range(num_layers)
        ])

        # 最后一层的输出会输入到全连接层进行分类
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size, time_steps, _, width, height = x.size()

        # 初始化细胞状态和隐藏状态
        h, c = [torch.zeros(batch_size, self.hidden_size, width, height).to(x.device)↙
for _ in range(2)]

        # 逐步处理每个时间步的数据
```

```
        for t in range(time_steps):
            x_t = x[:, t]  # 获取当前时间步的输入数据

            # 在每一层 ConvLSTM 上进行前向传播
            for layer in range(self.num_layers):
                h, c = self.layers[layer](x_t, h, c)

            # 当前时刻的输出
            x_t = h  # 当前时刻的隐藏状态作为下一个时间步的输入

        # 最后一层的输出经过全连接层分类
        x_t = x_t.view(batch_size, -1)  # 展平为一维
        out = self.fc(x_t)  # 预测输出

        return out


class ConvLSTM_Model(nn.Module):
    def __init__(self, input_channels=1, hidden_size=64, kernel_size=(3, 3),↙
num_layers=2, output_size=6):
        super(ConvLSTM_Model, self).__init__()

        # 定义ConvLSTM层
        self.conv_lstm = ConvLSTM(input_size=input_channels, hidden_size=hidden_size,
                                  kernel_size=kernel_size, num_layers=num_layers,↙
output_size=output_size)

    def forward(self, x):
        # x.shape = [batch_size, time_steps, channels, height, width]
        out = self.conv_lstm(x)
        return out


# # 测试模型结构
# model = ConvLSTM_Model(input_channels=1, hidden_size=64, kernel_size=(3, 3),↙
num_layers=2, output_size=6)
# input_tensor = torch.randn(32, 15, 1, 11, 11)  # 假设批量大小为32，15个时间步，1通道，↙
11x11空间
# output_tensor = model(input_tensor)
# print("Output shape:", output_tensor.shape)  # 输出应为 [32, 6]

#@title Attention
import torch
import torch.nn as nn
import torch.nn.functional as F

class TemporalAttention(nn.Module):
    def __init__(self, hidden_size):
        super(TemporalAttention, self).__init__()
        self.attn_weights = nn.Parameter(torch.randn(hidden_size, 1))  # 用于计算每个时间↙
步的注意力权重

    def forward(self, lstm_out):
        # lstm_out shape = [batch_size, time_steps, hidden_size]
        attn_scores = torch.bmm(lstm_out, self.attn_weights.unsqueeze(0).expand↙
```

```
(lstm_out.size(0), -1, -1))  # (batch_size, time_steps, 1)
        attn_weights = torch.softmax(attn_scores, dim=1)  # 计算每个时间步的注意力权重
        context = torch.sum(attn_weights * lstm_out, dim=1)  # 加权求和，得到时间步的上下文↙
向量
        return context

class CNN_LSTM_Attention_Model(nn.Module):
    def __init__(self, map_type):
        super(CNN_LSTM_Attention_Model, self).__init__()

        # 卷积层
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=0, stride=2)  # 每个时间步↙
输入1通道，输出32通道
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=0, stride=2)  # 输入32通↙
道，输出64通道

        # BatchNorm层
        self.bn1 = nn.BatchNorm2d(32)
        self.bn2 = nn.BatchNorm2d(64)

        # Dropout层
        self.dropout_conv = nn.Dropout2d(0.5)  # 卷积层之后的Dropout

        # 计算展平后的维度: 64 * 2 * 2
        self.flat_dim = 64 * 2 * 2

        # LSTM层，用于处理序列数据
        self.lstm = nn.LSTM(input_size=self.flat_dim, hidden_size=128, num_layers=1,↙
batch_first=True)

        # 时间注意力层
        self.attn = TemporalAttention(128)

        # 全连接层
        if map_type == "LULC":
            self.fc = nn.Linear(128, 8)  # 将LSTM的最后一个时间步的输出映射到8个分类
        else:
            self.fc = nn.Linear(128, 8)  # 将LSTM的最后一个时间步的输出映射到6个分类

    def forward(self, x):
        batch_size, time_steps, width, height = x.shape  # x.shape = [8192, 15, 11,↙
11]

        # 将时间步合并到批次维度，进行批量卷积操作
        x = x.view(batch_size * time_steps, 1, width, height)  # shape = [batch_size *↙
time_steps, 1, 11, 11]

        # CNN提取空间特征
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = self.dropout_conv(x)

        # 展平
        x = x.view(batch_size, time_steps, -1)  # shape = [batch_size, time_steps,↙
flat_dim]
```

```
        # 使用LSTM处理序列数据
        lstm_out, _ = self.lstm(x)  # lstm_out shape = [batch_size, time_steps, 128]

        # 使用时间注意力机制提取加权的序列特征
        x = self.attn(lstm_out)  # shape = [batch_size, hidden_size]

        # 全连接层进行分类
        x = self.fc(x)  # shape = [batch_size, 6]

        return x

# # 测试模型结构
# model = CNN_LSTM_Attention_Model(map_type=map_type)
# input_tensor = torch.randn(32, 15, 11, 11)  # 假设批量大小为32，15个时间步，11x11的空间
# output_tensor = model(input_tensor)
# # print("Output shape:", output_tensor.shape)  # 输出应为 [32, 6]

#@title ViT_Trans
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset

# Model parameters
patch_size = 11  # Each 11x11 grid is a single patch
emb_size = 64    # Embedding size
seq_len = 15     # Number of time steps
num_heads = 4    # Number of attention heads
num_layers = 2   # Number of transformer layers

# ViT + Transformer 模型定义
class ViTTransformer(nn.Module):
    def __init__(self, map_type, patch_size=11, emb_size=64, seq_len=15, num_heads=4,↙
num_layers=2):
        super(ViTTransformer, self).__init__()

        # Patch Embedding for each 11x11 grid (treated as one patch here)
        self.patch_size = patch_size
        self.emb_size = emb_size
        self.patch_embedding = nn.Linear(patch_size * patch_size, emb_size)

        # Learnable positional encoding for time steps
        self.positional_encoding = nn.Parameter(torch.randn(1, seq_len, emb_size))

        # Transformer Encoder for temporal modeling
        self.transformer = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(d_model=emb_size, nhead=num_heads,↙
batch_first=True),
            num_layers=num_layers
        )

        # Classification head

        if map_type == "LULC":
```

```
            self.fc = nn.Linear(emb_size, 8)  # 映射到8个分类
        else:
            self.fc = nn.Linear(emb_size, 8)  # 映射到6个分类

    def forward(self, x):
        # x: (B, T, H, W) -> (B, T, P)
        B, T, H, W = x.size()
        assert H == self.patch_size and W == self.patch_size, "Input size mismatch↙
with patch size"

        # Flatten each patch (11x11 -> 121)
        x = x.view(B, T, -1)  # (B, T, P)

        # Patch embedding (121 -> emb_size)
        x = self.patch_embedding(x)  # (B, T, emb_size)

        # Add positional encoding (T -> seq_len)
        x = x + self.positional_encoding[:, :T, :]  # (B, T, emb_size)

        # Transformer Encoder
        x = self.transformer(x)  # (B, T, emb_size)

        # Take the last time step for classification
        x = x[:, -1, :]  # (B, emb_size)

        # Classification
        output = self.fc(x)  # (B, num_classes)
        return output


# model = ViTTransformer(map_type=map_type)

#@title ViT_LSTM
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset

# Model parameters
patch_size = 11  # Each 11x11 grid is a single patch
emb_size = 64    # Embedding size
seq_len = 15     # Number of time steps
lstm_hidden_size = 128  # LSTM hidden size

# ViT + LSTM 模型定义
class ViTLSTM(nn.Module):
    def __init__(self, map_type, patch_size=11, emb_size=64, seq_len=15,↙
lstm_hidden_size=128):
        super(ViTLSTM, self).__init__()

        # Patch Embedding for each 11x11 grid (treated as one patch here)
        self.patch_size = patch_size
        self.emb_size = emb_size
        self.patch_embedding = nn.Linear(patch_size * patch_size, emb_size)
```

```
        # LSTM for temporal modeling
        self.lstm = nn.LSTM(emb_size, lstm_hidden_size, batch_first=True)

        # Classification head
        if map_type == "LULC":
            self.fc = nn.Linear(lstm_hidden_size, 8)  # 映射到8个分类
        else:
            self.fc = nn.Linear(lstm_hidden_size, 6)  # 映射到6个分类

    def forward(self, x):
        # x: (B, T, H, W) -> (B, T, P)
        B, T, H, W = x.size()
        assert H == self.patch_size and W == self.patch_size, "Input size mismatch↙
with patch size"

        # Flatten each patch (11x11 -> 121)
        x = x.view(B, T, -1)  # (B, T, P)

        # Patch embedding (121 -> emb_size)
        x = self.patch_embedding(x)  # (B, T, emb_size)

        # LSTM for temporal modeling
        x, (hn, cn) = self.lstm(x)  # (B, T, lstm_hidden_size)

        # Take the last time step for classification (or use the final hidden state)
        x = x[:, -1, :]  # (B, lstm_hidden_size)

        # Classification
        output = self.fc(x)  # (B, num_classes)
        return output

# # Example usage
# model = ViTLSTM(map_type=map_type)

# # Example input
# x = torch.randn(32, 15, 11, 11)  # Batch size 32, 15 time steps, 11x11 grid
# output = model(x)
# print(output.shape)  # Expected: torch.Size([32, 6])

#@title COA
import numpy as np

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

class CoatiOptimization:
    def __init__(self, model, objective_function, param_bounds, population_size=20,↙
max_iterations=50):
        """
        初始化Coati优化算法
        :param model: 待优化的模型
        :param objective_function: 评估模型性能的目标函数
        :param param_bounds: 参数边界，字典形式 {param_name: (min_value, max_value)}
        :param population_size: 水獭个体数量
        :param max_iterations: 最大迭代次数
        """
```

```
        self.model = model
        self.objective_function = objective_function
        self.param_bounds = param_bounds
        self.population_size = population_size
        self.max_iterations = max_iterations
        self.population = self.initialize_population()

    def initialize_population(self):
        """随机初始化水獭群体参数"""
        population = []
        for _ in range(self.population_size):
            individual = {param: np.random.uniform(low, high) for param, (low, high)↙
in self.param_bounds.items()}
            population.append(individual)
        return population

    def optimize(self):
        """执行Coati优化算法"""
        for iteration in range(self.max_iterations):
            # 评估每个水獭个体的适应度
            fitness = [self.objective_function(self.model, individual) for individual↙
in self.population]
            # 根据适应度排序
            sorted_population = [x for _, x in sorted(zip(fitness, self.population),↙
key=lambda pair: pair[0])]

            # 更新水獭的参数（模拟捕食行为）
            best_individual = sorted_population[0]
            for i, individual in enumerate(self.population):
                for param in individual:
                    # 模拟水獭追捕猎物的行为（简单调整参数）
                    step = (best_individual[param] - individual[param]) * np.random.↙
rand()
                    individual[param] += step
                    # 确保参数在边界内
                    individual[param] = np.clip(individual[param], *self.param_bounds↙
[param])
        return best_individual


def objective_function(model, params):
    """
    用于优化的目标函数。

    参数:
    - params: 字典，包含需要优化的超参数。

    返回:
    - 验证集损失或指标值（如准确率）。
    """
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,↙
pin_memory=True)
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,↙
pin_memory=True)
```

```
    # 模型初始化
    model = update_model_with_params(model, params)
    model = model.to(device)

    # 定义损失函数和优化器
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

    # 训练模型
    num_epochs = 5
    for epoch in range(num_epochs):
        model.train()
        train_loss = 0.0
        for inputs_batch, labels_batch in tqdm(train_loader, desc=f"train {epoch}",↙
leave=False):
            inputs_batch, labels_batch = inputs_batch.to(device), labels_batch.to↙
(device)

            # 清空梯度
            optimizer.zero_grad()

            # 前向传播
            outputs = model(inputs_batch)

            # 计算损失
            loss = criterion(outputs, labels_batch)

            # 反向传播和优化
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

    # 验证模型
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs_batch, labels_batch in tqdm(val_loader, desc=f"val_{epoch}",↙
leave=False):
            inputs_batch, labels_batch = inputs_batch.to(device), labels_batch.to↙
(device)

            # 前向传播
            outputs = model(inputs_batch)
            loss = criterion(outputs, labels_batch)
            val_loss += loss.item()

            # 计算准确率
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels_batch).sum().item()
            total += labels_batch.size(0)

    val_loss /= len(val_loader)
```

```
    accuracy = correct / total

    print(f"Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.4f}")

    # 返回验证集损失（用于最小化）
    return val_loss



def update_model_with_params(model, best_params):
    """
    根据最佳参数更新模型结构
    :param model: 原始模型实例
    :param best_params: Coati优化得到的最佳参数
    :return: 更新后的模型
    """
    # 更新CNN层
    model.conv1 = nn.Conv2d(
        1,
        int(best_params['conv1_out_channels']),
        kernel_size=int(best_params['kernel_size']),
        stride=2
    )
    model.conv2 = nn.Conv2d(
        int(best_params['conv1_out_channels']),
        int(best_params['conv2_out_channels']),
        kernel_size=int(best_params['kernel_size']),
        stride=2
    )

    # 更新展平维度计算 (需要重新推断卷积后的形状)
    dummy_input = torch.randn(1, 1, 11, 11)  # 假设输入为[batch_size, channel, height,↙
width]
    dummy_output = model.conv2(model.conv1(dummy_input))
    flat_dim = int(torch.prod(torch.tensor(dummy_output.shape[1:])))  # 计算展平后的维度

    # 更新LSTM层
    model.flat_dim = flat_dim
    model.lstm = nn.LSTM(
        input_size=model.flat_dim,
        hidden_size=int(best_params['lstm_hidden_size']),
        num_layers=1,
        batch_first=True
    )

    # 更新全连接层的输入大小为LSTM的隐藏层大小
    model.fc = nn.Linear(
        in_features=int(best_params['lstm_hidden_size']),
        out_features=model.fc.out_features  # 分类数保持不变
    )

    return model
```

```
# param_bounds = {
#     'conv1_out_channels': (16, 64),
#     'conv2_out_channels': (32, 128),
#     'kernel_size': (2, 5),
#     'lstm_input_size': (256, 1024),
#     'lstm_hidden_size': (64, 256)
# }


# # 创建模型实例


# # 初始化Coati优化算法
# coati_optimizer = CoatiOptimization(
#     model=model,
#     objective_function=objective_function,
#     param_bounds=param_bounds,
#     population_size=10,
#     max_iterations=10
# )


# # 优化超参数
# best_params = coati_optimizer.optimize()


# # 使用最佳参数更新模型
# print("Best parameters:", best_params)

# model = update_model_with_params(model, best_params)

"""# Train"""

#@title AMP Train

import torch.optim as optim
import torch.nn.functional as F
import os
from torch.optim.lr_scheduler import ReduceLROnPlateau
from torch.amp import GradScaler, autocast
from tqdm import tqdm

model_dict = {
    "CNN": PureCNNModel(map_type=map_type),
    "LSTM": PureLSTMModel(map_type=map_type),
    "CNNLSTM": CNN_LSTM_Model(map_type=map_type),
    "Attention": CNN_LSTM_Attention_Model(map_type=map_type),
    "ViT_Trans": ViTTransformer(map_type=map_type),
    "ViT_LSTM": ViTLSTM(map_type=map_type)
}

model = model_dict[model_type]

# 设置训练超参数
```

```
epochs = 500  # 训练的轮数
learning_rate = 0.001
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
batch_size = 4096 * 4 * 2

num_workers = 2
train_loader = DataLoader(train_dataset, batch_size=batch_size, sampler=sampler,↙
shuffle=False, pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,↙
pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,↙
pin_memory=True)

# 损失函数和优化器
criterion = nn.CrossEntropyLoss()  # 适用于分类任务

# 将模型移动到GPU (如果可用)
print(device)
model.to(device)

optimizer = optim.Adam(model.parameters(), lr=learning_rate)
scheduler = ReduceLROnPlateau(optimizer, 'min', patience=5, factor=0.1)

# 初始化 GradScaler 用于 AMP
scaler = GradScaler('cuda')

# 训练模型
for epoch in range(epochs):
    model.train()  # 设置模型为训练模式
    running_loss = 0.0
    correct = 0
    total = 0

    # 遍历训练数据
    for inputs_batch, labels_batch in tqdm(train_loader, desc="Training...",↙
leave=False):
        inputs_batch, labels_batch = inputs_batch.to(device), labels_batch.to(device)
        optimizer.zero_grad()

        # 使用 AMP 进行前向和反向传播
        with autocast('cuda'):
            outputs = model(inputs_batch)
            loss = criterion(outputs, labels_batch)

        # Scaler 处理梯度缩放
        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()

        # 记录训练损失和准确率
        running_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total += labels_batch.size(0)
        correct += (predicted == labels_batch).sum().item()
```

```
    # 计算训练集的平均损失和准确度
    epoch_loss = running_loss / len(train_loader)
    epoch_accuracy = 100 * correct / total

    # 验证集上的损失和准确度
    model.eval()  # 切换为评估模式
    val_loss = 0.0
    val_correct = 0
    val_total = 0

    with torch.no_grad():  # 禁用梯度
        for val_inputs, val_labels in tqdm(val_loader, desc="validating...",↙
leave=False):
            val_inputs, val_labels = val_inputs.to(device), val_labels.to(device)
            with autocast('cuda'):  # 在验证阶段也使用 AMP
                val_outputs = model(val_inputs)
                val_loss += criterion(val_outputs, val_labels).item()

            _, val_predicted = torch.max(val_outputs, 1)
            val_total += val_labels.size(0)
            val_correct += (val_predicted == val_labels).sum().item()

    avg_val_loss = val_loss / len(val_loader)
    val_accuracy = 100 * val_correct / val_total

    # 更新学习率调度器
    scheduler.step(avg_val_loss)
    last_lr = scheduler.get_last_lr()

    # 保存训练日志
    os.makedirs("models", exist_ok=True)
    with open("models/training_log.txt", "a") as log_file:
        log_message = (f"Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss:.4f}, "
                       f"Accuracy: {epoch_accuracy:.2f}%, Val Loss: {avg_val_loss:.↙
4f}, "
                       f"Val Accuracy: {val_accuracy:.2f}%,  Learning rate: {last_lr}↙
\n")
        print(log_message, end='')  # 控制台输出
        log_file.write(log_message)  # 写入文件

    # 每个epoch后保存模型
    if epoch % 10 == 0:
        os.makedirs(f"models/saves/{map_type}/", exist_ok=True)
        model_save_path = f"models/saves/{map_type}/{model_type}_{epoch+1}_↙
{val_accuracy:.2f}%.pth"
        torch.save(model.state_dict(), model_save_path)

#@title Train
import torch.optim as optim
import torch.nn.functional as F
import os
from torch.optim.lr_scheduler import ReduceLROnPlateau

# 设置训练超参数
epochs = 500  # 训练的轮数
```

```
learning_rate = 0.01
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# batch_size = 1024
batch_size = 4096 * 4
num_workers = 2
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True,↙
pin_memory=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False,↙
pin_memory=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,↙
pin_memory=True)

# 损失函数和优化器
criterion = nn.CrossEntropyLoss()  # 适用于分类任务

# 将模型移动到GPU (如果可用)
model.to(device)

optimizer = optim.Adam(model.parameters(), lr=learning_rate)
scheduler = ReduceLROnPlateau(optimizer, 'min', patience=3, factor=0.1)

# 训练模型
for epoch in range(epochs):
    model.train()  # 设置模型为训练模式
    running_loss = 0.0
    correct = 0
    total = 0

    # 遍历训练数据
    for inputs_batch, labels_batch in tqdm(train_loader, desc="Training...",↙
leave=False):
        inputs_batch, labels_batch = inputs_batch.to(device), labels_batch.to(device)
        optimizer.zero_grad()
        outputs = model(inputs_batch)
        # print(inputs_batch.shape, outputs.shape)
        loss = criterion(outputs, labels_batch)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total += labels_batch.size(0)
        correct += (predicted == labels_batch).sum().item()

    # 计算训练集的平均损失和准确度
    epoch_loss = running_loss / len(train_loader)
    epoch_accuracy = 100 * correct / total

    # 验证集上的损失和准确度
    model.eval()  # 切换为评估模式
    val_loss = 0.0
    val_correct = 0
    val_total = 0

    with torch.no_grad():  # 禁用梯度
        for val_inputs, val_labels in tqdm(val_loader, desc="validating...",↙
```

```
leave=False):
            val_inputs, val_labels = val_inputs.to(device), val_labels.to(device)
            val_outputs = model(val_inputs)
            val_loss += criterion(val_outputs, val_labels).item()

            _, val_predicted = torch.max(val_outputs, 1)
            val_total += val_labels.size(0)
            val_correct += (val_predicted == val_labels).sum().item()

    avg_val_loss = val_loss / len(val_loader)
    val_accuracy = 100 * val_correct / val_total
    # print(f"Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss:.4f}, Accuracy:↙
{epoch_accuracy:.2f}%, Val Loss: {avg_val_loss:.4f}, Val Accuracy: {val_accuracy:.↙
2f}%")

    # 更新学习率调度器
    scheduler.step(avg_val_loss)
    last_lr = scheduler.get_last_lr()

    with open("models/training_log.txt", "a") as log_file:  # 使用 "a" 模式追加写入
        # log_message = f"Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss:.4f},↙
Accuracy: {epoch_accuracy:.2f}%, Val Loss: {avg_val_loss:.4f}, Val Accuracy:↙
{val_accuracy:.2f}%\n"
        log_message = f"Epoch [{epoch+1}/{epochs}], Loss: {epoch_loss:.4f}, Accuracy:↙
{epoch_accuracy:.2f}%, Val Loss: {avg_val_loss:.4f}, Val Accuracy: {val_accuracy:.↙
2f}%， Learning rate: {last_lr}\n"
        print(log_message, end='')  # 控制台输出
        log_file.write(log_message)  # 写入文件


    # 每个epoch后保存模型
    os.makedirs("models", exist_ok=True)
    model_save_path = f"models/{epoch+1}.pth"
    torch.save(model.state_dict(), model_save_path)
    # print(f"Model saved at {model_save_path}")

"""# Test"""

#@title Test
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import os
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,↙
roc_auc_score, confusion_matrix, classification_report


model_dict = {
    "CNN": PureCNNModel(map_type=map_type),
    "LSTM": CNN_LSTM_Attention_Model(map_type=map_type),
    "CNNLSTM": CNN_LSTM_Model(map_type=map_type),
    "Attention": CNN_LSTM_Attention_Model(map_type=map_type),
    "ViT_Trans": ViTTransformer(map_type=map_type),
    "ViT_LSTM": ViTLSTM(map_type=map_type)
}
```

```
model = model_dict[model_type]

model.load_state_dict(torch.load(f'models/{map_type}/{model_type}.pth',↙
weights_only=True, map_location=torch.device('cpu')))

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

batch_size = 4096 * 4
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,↙
pin_memory=True)

all_targets = []
all_outputs = []

with torch.no_grad():
  for inputs, targets in tqdm(test_loader):
    inputs = inputs.to(device)
    targets = targets.to(device)
    # print(inputs.shape)
    outputs = model(inputs)
    # print(outputs.shape)
    outputs = np.argmax(outputs.cpu(), axis=1)
    targets = targets.cpu()
    # print(inputs.shape, outputs.shape)

    outputs = outputs + 1
    targets = targets + 1

    all_targets.extend(targets.numpy())
    all_outputs.extend(outputs.numpy())


plt.rcParams['font.family'] = 'Serif'
fontsize = 14
title_fontsize = 16

# 1. 准确率 (Accuracy)
accuracy = accuracy_score(all_targets, all_outputs)
print(f"Accuracy: {accuracy:.4f}")

# 2. 精确率 (Precision) -  每个类别的精确率
precision_macro = precision_score(all_targets, all_outputs, average='macro')
precision_micro = precision_score(all_targets, all_outputs, average='micro')
precision_weighted = precision_score(all_targets, all_outputs, average='weighted')
print(f"Macro Precision: {precision_macro:.4f}")
print(f"Micro Precision: {precision_micro:.4f}")
print(f"Weighted Precision: {precision_weighted:.4f}")

# 3. 召回率 (Recall) - 每个类别的召回率
recall_macro = recall_score(all_targets, all_outputs, average='macro')
recall_micro = recall_score(all_targets, all_outputs, average='micro')
recall_weighted = recall_score(all_targets, all_outputs, average='weighted')
print(f"Macro Recall: {recall_macro:.4f}")
```

```
print(f"Micro Recall: {recall_micro:.4f}")
print(f"Weighted Recall: {recall_weighted:.4f}")

# 4. F1 分数 (F1-score) - 每个类别的F1分数，是精确率和召回率的调和平均数
f1_macro = f1_score(all_targets, all_outputs, average='macro')
f1_micro = f1_score(all_targets, all_outputs, average='micro')
f1_weighted = f1_score(all_targets, all_outputs, average='weighted')
print(f"Macro F1-score: {f1_macro:.4f}")
print(f"Micro F1-score: {f1_micro:.4f}")
print(f"Weighted F1-score: {f1_weighted:.4f}")

# 5. 混淆矩阵 (Confusion Matrix)
cm = confusion_matrix(all_targets, all_outputs)
# print("Confusion Matrix:\n", cm)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

labels = [1, 2, 3, 4, 5, 6, 7, 8]
labels = [1, 2, 3, 4, 5, 6]

# 设置图形大小
plt.figure(figsize=(10, 8))

# 使用seaborn绘制热力图
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=labels,↙
yticklabels=labels) # annot=True 显示数值, fmt='d' 保持整数显示

# 添加标签和标题
plt.xlabel('Predicted Label', fontsize=fontsize)
plt.ylabel('True Label', fontsize=fontsize)
plt.title('Confusion Matrix', fontsize=title_fontsize)

# # tick_marks = np.arange(cm.shape[0]) + 1
# plt.xticks(tick_marks, tick_marks)
# plt.yticks(tick_marks, tick_marks)

# 显示图形
os.makedirs(f'outputs/plots/{model_type}/matrix/', exist_ok=True)
plt.savefig(f'outputs/plots/{model_type}/matrix/{map_type}_confusion_matrix.png')
plt.show()

# 绘制百分比混淆矩阵
plt.figure(figsize=(10, 8))
sns.heatmap(cm_normalized, annot=True, fmt=".2f", cmap="Blues", cbar=False,↙
xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted Label', fontsize=fontsize)
plt.ylabel('True Label', fontsize=fontsize)
plt.title('Normalized Confusion Matrix', fontsize=title_fontsize)


# # tick_marks = np.arange(cm.shape[0]) + 1
# plt.xticks(tick_marks, tick_marks)
# plt.yticks(tick_marks, tick_marks)

plt.savefig(f'outputs/plots/{model_type}/matrix/{map_type}↙
_normalized_confusion_matrix.png')
```

```
plt.show()

# 6. 分类报告 (Classification Report) - 包含精确率，召回率，F1分数
report = classification_report(all_targets, all_outputs)
print("Classification Report:\n", report)


from sklearn.preprocessing import label_binarize
from sklearn.metrics import precision_recall_curve, average_precision_score

plt.rcParams['font.family'] = 'Serif'
fontsize = 14

fvc_class_map = {
    0: None,  # 空
    1: "Low vegetation coverage",  # L
    2: "Relatively low vegetation coverage",  # RL
    3: "Moderate vegetation coverage",  # M
    4: "Relatively high vegetation coverage",  # RH
    5: "High vegetation coverage",  # H
    6: "blank",  # 白色像素
}

lulc_class_map = {
    0: None,  # 空
    1: "Cropland",  # L
    2: "Forest",  # RL
    3: "Grassland",  # M
    4: "Water",  # RH
    5: "Impervious",  # H
    6: "Barren",  # 白色像素
    7: "Snow/Ice",  # 白色像素
    8: "blank",  # 白色像素
}

rsei_class_map = {
    0: None,  # 空
    1: "Poor",  # L
    2: "Fair",  # RL
    3: "Moderate",  # M
    4: "Good",  # RH
    5: "Excellent",  # H
    6: "blank",  # 白色像素
}

map_type_map = {
    'FVC': fvc_class_map,
    'LULC': lulc_class_map,
    'RSEI': rsei_class_map,
}

# 假设类别数为 n_classes
n_classes = len(np.unique(all_targets))

# 将目标值进行二值化（one-hot 编码）
```

```
y_bin = label_binarize(all_targets, classes=np.arange(n_classes))
y_pred_bin = label_binarize(all_outputs, classes=np.arange(n_classes))

# 计算每个类别的 Precision-Recall 曲线和 Average Precision
precision = dict()
recall = dict()
average_precision = dict()
for i in range(1, n_classes):
    precision[i], recall[i], _ = precision_recall_curve(y_bin[:, i], y_pred_bin[:, i])
    average_precision[i] = average_precision_score(y_bin[:, i], y_pred_bin[:, i])

# 绘制每个类别的 Precision-Recall 曲线
plt.figure(figsize=(10, 8))
for i in range(1, n_classes):
    plt.plot(recall[i], precision[i], label=f'{map_type_map[map_type][i]} (AP =↙
{average_precision[i]:.2f})')

plt.xlabel('Recall', fontsize=fontsize)
plt.ylabel('Precision', fontsize=fontsize)
plt.title('Precision-Recall Curve', fontsize=title_fontsize)
plt.legend(loc='best', fontsize=fontsize)
plt.grid()
os.makedirs(f'outputs/plots/{model_type}/PR/', exist_ok=True)
plt.savefig(f'outputs/plots/{model_type}/PR/{map_type}_PR_curve.png')
plt.show()


from sklearn.metrics import roc_curve, auc

# 计算每个类别的 ROC 曲线和 AUC
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(1, n_classes):
    fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_pred_bin[:, i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# 绘制每个类别的 ROC 曲线
plt.figure(figsize=(10, 8))
for i in range(1, n_classes):
    plt.plot(fpr[i], tpr[i], label=f'{map_type_map[map_type][i]} (AUC = {roc_auc[i]:.↙
2f})')

plt.plot([0, 1], [0, 1], 'k--', label='Random Guess')
plt.xlabel('False Positive Rate', fontsize=fontsize)
plt.ylabel('True Positive Rate', fontsize=fontsize)
plt.title('ROC Curve', fontsize=title_fontsize)
plt.legend(loc='best', fontsize=fontsize)
plt.grid()
os.makedirs(f'outputs/plots/{model_type}/ROC/', exist_ok=True)
plt.savefig(f'outputs/plots/{model_type}/ROC/{map_type}_ROC_curve.png')
plt.show()

#@title Test All
import matplotlib.pyplot as plt
```

```
import numpy as np
import seaborn as sns
import os
import pandas as pd
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,↙
roc_auc_score, confusion_matrix, classification_report
import matplotlib.patches as mpatches
from sklearn.preprocessing import label_binarize
from sklearn.metrics import precision_recall_curve, average_precision_score

plt.rcParams['font.family'] = 'Serif'
# fontsize = 14

fvc_class_map = {
    0: None,  # 空
    1: "Low",  # L
    2: "Relatively low",  # RL
    3: "Moderate",  # M
    4: "Relatively high",  # RH
    5: "High",  # H
    6: "blank",  # 白色像素
}

lulc_class_map = {
    0: None,  # 空
    1: "Cropland",  # L
    2: "Forest",  # RL
    3: "Grassland",  # M
    4: "Water",  # RH
    5: "Impervious",  # H
    6: "Barren",  # 白色像素
    7: "Snow/Ice",  # 白色像素
    8: "blank",  # 白色像素
}

rsei_class_map = {
    0: None,  # 空
    1: "Poor",  # L
    2: "Fair",  # RL
    3: "Moderate",  # M
    4: "Good",  # RH
    5: "Excellent",  # H
    6: "blank",  # 白色像素
}

map_type_map = {
    'FVC': fvc_class_map,
    'LULC': lulc_class_map,
    'RSEI': rsei_class_map,
}

model_dict = {
    "CNN": PureCNNModel(map_type=map_type),
    "LSTM": PureLSTMModel(map_type=map_type),
    # "LSTM": CNN_LSTM_Attention_Model(map_type=map_type),
```

```
    "CNNLSTM": CNN_LSTM_Model(map_type=map_type),
    "Attention": CNN_LSTM_Attention_Model(map_type=map_type),
    "ViT_Trans": ViTTransformer(map_type=map_type),
    "ViT_LSTM": ViTLSTM(map_type=map_type)
}

for model_type in ['CNN', 'LSTM', 'CNNLSTM', 'Attention'][:]:
  # print(f"Testing {map} - {model}")
  print(f"Testing {map_type} - {model_type}")
  model = model_dict[model_type]
  model.load_state_dict(torch.load(f'models/{map_type}/{model_type}.pth',↙
weights_only=True, map_location=torch.device('cpu')))
  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
  model.to(device)

  batch_size = 4096 * 4
  test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,↙
pin_memory=True)

  all_targets = []
  all_outputs = []

  with torch.no_grad():
    for inputs, targets in tqdm(test_loader):
      inputs = inputs.to(device)
      targets = targets.to(device)
      # print(inputs.shape)
      outputs = model(inputs)
      # print(outputs.shape)
      outputs = outputs[:, :-1]  # 去除白色标签概率
      outputs = np.argmax(outputs.cpu(), axis=1)
      targets = targets.cpu()

      all_targets.extend(targets.numpy())
      all_outputs.extend(outputs.numpy())


  plt.rcParams['font.family'] = 'Serif'
  # title_fontsize = 24
  # fontsize = 22
  # tick_fontsize = 22
  # labelsize = 18
  title_fontsize = 32  # 标题
  legend_fontsize = 28  # 图例
  tick_fontsize = 28  # 坐标刻度
  axis_fontsize = 32  # 坐标轴标题
  matrix_value_fontsize = 28  # 热力图值标签

  with_legend = True

  # 1. 准确率 (Accuracy)
  accuracy = accuracy_score(all_targets, all_outputs)

  # 2. 精确率 (Precision) - 每个类别的精确率
  precision_macro = precision_score(all_targets, all_outputs, average='macro')
```

```
  precision_micro = precision_score(all_targets, all_outputs, average='micro')
  precision_weighted = precision_score(all_targets, all_outputs, average='weighted')

  # 3. 召回率 (Recall) - 每个类别的召回率
  recall_macro = recall_score(all_targets, all_outputs, average='macro')
  recall_micro = recall_score(all_targets, all_outputs, average='micro')
  recall_weighted = recall_score(all_targets, all_outputs, average='weighted')

  # 4. F1 分数 (F1-score) -  每个类别的F1分数，是精确率和召回率的调和平均数
  f1_macro = f1_score(all_targets, all_outputs, average='macro')
  f1_micro = f1_score(all_targets, all_outputs, average='micro')
  f1_weighted = f1_score(all_targets, all_outputs, average='weighted')

  # print(f"Accuracy: {accuracy:.4f}")
  # print()
  # print(f"Macro Precision: {precision_macro:.4f}")
  # print()
  # print(f"Micro Precision: {precision_micro:.4f}")
  # print()
  # print(f"Weighted Precision: {precision_weighted:.4f}")
  # print()
  # print(f"Macro Recall: {recall_macro:.4f}")
  # print()
  # print(f"Micro Recall: {recall_micro:.4f}")
  # print()
  # print(f"Weighted Recall: {recall_weighted:.4f}")
  # print()
  # print(f"Macro F1-score: {f1_macro:.4f}")
  # print()
  # print(f"Micro F1-score: {f1_micro:.4f}")
  # print()
  # print(f"Weighted F1-score: {f1_weighted:.4f}")

  # 5. 混淆矩阵 (Confusion Matrix)
  cm = confusion_matrix(all_targets, all_outputs)
  cm = cm[:, :-1]

  # print("Confusion Matrix:\n", cm)
  cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]


  xlabels = list(range(0, len(np.unique(all_targets))-1))
  ylabels = list(range(0, len(np.unique(all_targets))))

  # 设置图形大小
  plt.figure(figsize=(10, 8))

  # 使用seaborn绘制热力图
  sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False, xticklabels=xlabels,↙
yticklabels=ylabels, annot_kws={"size": matrix_value_fontsize}) # annot=True 显示数值,↙
fmt='d' 保持整数显示

  # 添加标签和标题
  plt.xlabel('Predicted Label', fontsize=axis_fontsize)
  plt.ylabel('True Label', fontsize=axis_fontsize)
```

```
  plt.title('Confusion Matrix', fontsize=title_fontsize)
  legend_labels = [mpatches.Patch(color='white', label=f'{i}: {map_type_map[map_type]↙
[i+1]}') for i in ylabels]

  # plt.legend(handles=legend_labels, loc='upper right', title='Labels', fontsize=10,↙
title_fontsize=10)
  if with_legend:
    plt.legend(handles=legend_labels, bbox_to_anchor=(1.8, 0.9), title='Labels',↙
fontsize=legend_fontsize, title_fontsize=legend_fontsize)

  plt.xticks(fontsize=tick_fontsize)
  plt.yticks(fontsize=tick_fontsize)

  # 显示图形
  os.makedirs(f'outputs/plots/{model_type}/matrix/', exist_ok=True)
  plt.savefig(f'outputs/plots/{model_type}/matrix/{map_type}_confusion_matrix.png',↙
bbox_inches='tight')
  # plt.show()

  # 绘制百分比混淆矩阵
  plt.figure(figsize=(10, 8))
  sns.heatmap(cm_normalized, annot=True, fmt=".2f", cmap="Blues", cbar=False,↙
xticklabels=xlabels, yticklabels=ylabels, annot_kws={"size": matrix_value_fontsize})
  plt.xlabel('Predicted Label', fontsize=axis_fontsize)
  plt.ylabel('True Label', fontsize=axis_fontsize)
  plt.title('Normalized Confusion Matrix', fontsize=title_fontsize)
  if with_legend:
    plt.legend(handles=legend_labels, bbox_to_anchor=(1.8, 0.9), title='Labels',↙
fontsize=legend_fontsize, title_fontsize=legend_fontsize)


  # # tick_marks = np.arange(cm.shape[0]) + 1
  plt.xticks(fontsize=tick_fontsize)
  plt.yticks(fontsize=tick_fontsize)

  plt.savefig(f'outputs/plots/{model_type}/matrix/{map_type}↙
_normalized_confusion_matrix.png', bbox_inches='tight')
  # plt.show()

  # 6. 分类报告 (Classification Report) - 包含精确率，召回率，F1分数
  report = classification_report(all_targets, all_outputs, output_dict=True)
  report_df = pd.DataFrame(report).transpose()
  # print("Classification Report:\n", report)
  os.makedirs(f'outputs/plots/{model_type}/Report/', exist_ok=True)
  report_df.to_csv(f'outputs/plots/{model_type}/Report/{map_type}_Report.csv',↙
index=True)


  # 假设类别数为 n_classes
  n_classes = len(np.unique(all_targets))-1

  # 将目标值进行二值化（one-hot 编码）
  y_bin = label_binarize(all_targets, classes=np.arange(n_classes))
  y_pred_bin = label_binarize(all_outputs, classes=np.arange(n_classes))
```

```
  # 计算每个类别的 Precision-Recall 曲线和 Average Precision
  precision = dict()
  recall = dict()
  average_precision = dict()
  for i in range(0, n_classes):
      precision[i], recall[i], _ = precision_recall_curve(y_bin[:, i], y_pred_bin[:,
i])
      average_precision[i] = average_precision_score(y_bin[:, i], y_pred_bin[:, i])

  # 绘制每个类别的 Precision-Recall 曲线
  plt.figure(figsize=(10, 8))
  for i in range(0, n_classes):
      # plt.plot(recall[i], precision[i], label=f'{map_type_map[map_type][i+1]} (AP =
{average_precision[i]:.2f})')
      plt.plot(recall[i], precision[i], label=f'{map_type_map[map_type][i+1]}')

  plt.xlabel('Recall', fontsize=axis_fontsize)
  plt.ylabel('Precision', fontsize=axis_fontsize)
  plt.tick_params(axis='x', labelsize=tick_fontsize)
  plt.tick_params(axis='y', labelsize=tick_fontsize)
  plt.title('Precision-Recall Curve', fontsize=title_fontsize)
  if with_legend:
    plt.legend(loc='best', fontsize=legend_fontsize, bbox_to_anchor=(1.6, 0.9))
  plt.grid()
  os.makedirs(f'outputs/plots/{model_type}/PR/', exist_ok=True)
  plt.savefig(f'outputs/plots/{model_type}/PR/{map_type}_PR_curve.png',
bbox_inches='tight')
  # plt.show()


  from sklearn.metrics import roc_curve, auc

  # 计算每个类别的 ROC 曲线和 AUC
  fpr = dict()
  tpr = dict()
  roc_auc = dict()
  for i in range(0, n_classes):
      fpr[i], tpr[i], _ = roc_curve(y_bin[:, i], y_pred_bin[:, i])
      roc_auc[i] = auc(fpr[i], tpr[i])

  # 绘制每个类别的 ROC 曲线
  plt.figure(figsize=(10, 8))
  for i in range(0, n_classes):
      # plt.plot(fpr[i], tpr[i], label=f'{map_type_map[map_type][i+1]} (AUC = {roc_auc
[i]:.2f})')
      plt.plot(fpr[i], tpr[i], label=f'{map_type_map[map_type][i+1]}')

  plt.plot([0, 1], [0, 1], 'k--', label='Random Guess')
  plt.xlabel('False Positive Rate', fontsize=axis_fontsize)
  plt.ylabel('True Positive Rate', fontsize=axis_fontsize)
  plt.tick_params(axis='x', labelsize=tick_fontsize)
  plt.tick_params(axis='y', labelsize=tick_fontsize)
  plt.title('ROC Curve', fontsize=title_fontsize)
  if with_legend:
    plt.legend(loc='best', fontsize=legend_fontsize, bbox_to_anchor=(1.6, 0.9))
```

```
  plt.grid()
  os.makedirs(f'outputs/plots/{model_type}/ROC/', exist_ok=True)
  plt.savefig(f'outputs/plots/{model_type}/ROC/{map_type}_ROC_curve.png',↙
bbox_inches='tight')
  # plt.show()

"""# Graph gen"""

# 导入必要的包
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model_dict = {
    "CNN": PureCNNModel(map_type=map_type),
    "LSTM": CNN_LSTM_Attention_Model(map_type=map_type),
    "CNNLSTM": CNN_LSTM_Model(map_type=map_type),
    "Attention": CNN_LSTM_Attention_Model(map_type=map_type),
    "ViT_Trans": ViTTransformer(map_type=map_type),
    "ViT_LSTM": ViTLSTM(map_type=map_type)
}

# 加载保存的模型
model = model_dict[model_type].to(device)
print(f'models/{map_type}/{model_type}.pth')
model.load_state_dict(torch.load(f'models/{map_type}/{model_type}.pth',↙
weights_only=True))

"""## data process"""

import numpy as np
from tqdm import tqdm

# 载入数据并提取最后15年的数据
fvc_data = np.load(f'{map_type}.npz')['arr_0']
mask = np.load('whole_mask.npz')['arr_0']
latest_15_years_data = fvc_data[-15:]  # 提取最后15年的数据，形状为 (15, 4416, 5786)

predicted_map = np.zeros(latest_15_years_data.shape[1:])  # 初始化一个数组来保存预测结果

mask = np.load('whole_mask.npz')['arr_0']

# 设置区域大小
region_size = 10
offset = region_size // 2
height, width = latest_15_years_data.shape[1:]


indices = np.nonzero(mask[offset:height - offset, offset:width - offset])
indices = list(zip(indices[0] + offset, indices[1] + offset))

"""## predict"""

import torch
```

```
model.eval()  # 切换模型到评估模式
# 批量大小
batch_size = 1024 * 8 * 2  # 可以尝试调整批量大小，根据设备内存情况选择合适的数值

# 初始化结果数组
predicted_map = np.zeros(latest_15_years_data.shape[1:], dtype=np.uint8)

# 将 indices 划分为批次
batches = [indices[i:i + batch_size] for i in range(0, len(indices), batch_size)]

# 遍历每个批次
for batch in tqdm(batches):
    # 构造批量输入
    input_batch = []
    for i, j in batch:
        input_data = latest_15_years_data[:, i - offset:i + offset + 1, j - offset:j +
offset + 1]
        input_batch.append(input_data)

    # 转换为 Tensor 并移动到设备
    input_tensor = torch.tensor(np.array(input_batch), dtype=torch.float32).to(device)
# 形状为 (batch_size, 15, 11, 11)

    # 执行批量预测
    with torch.no_grad():
        outputs = model(input_tensor)
        # 去除白色像素
        outputs = outputs[:, :-1]
        _, predicted_classes = torch.max(outputs, 1)  # 获取预测的类别
        # print(outputs)
        # print(outputs[:-1])

    # 将预测结果写入预测地图
    for (i, j), predicted_class in zip(batch, predicted_classes):
        predicted_map[i, j] = predicted_class.item() + 1  # 提取数值并写入最终结果

"""## show plot"""

from google.colab.patches import cv2_imshow
import cv2


fvc_color_map = {
    0: [0, 0, 0],  # 空
    1: [0, 56, 168],  # L
    2: [0, 115, 38],  # H
    3: [82, 142, 249],  # RL
    4: [103, 203, 134],  # RH
    5: [190, 255, 255],  # M
    6: [255, 255, 255],  # 白色像素
}

lulc_color_map = {
    0: [0, 0, 0],  # 空
    1: [115, 255, 255],  # Cropland
```

```
    2: [0, 168, 112],  # Forest
    3: [190, 255, 233],  # Grassland
    4: [230, 92, 0],  # Water
    5: [0, 76, 230],  # Impervious
    6: [52, 52, 52],  # Barren
    7: [204, 204, 204],  # Snow/Ice
    8: [255, 255, 255],  # 白色像素
}

rsei_color_map = {
    0: [0, 0, 0],  # 空
    1: [0, 0, 255],  # Poor
    2: [0, 128, 255],  # Fair
    3: [0, 255, 255],  # Moderate
    4: [0, 212, 141],  # Good
    5: [0, 168, 56],  # Excellent
    6: [255, 255, 255],  # 白色像素
}

map_type_map = {
    'FVC': fvc_color_map,
    'LULC': lulc_color_map,
    'RSEI': rsei_color_map,
}

def img_2d_to_3d_vector(int_image, map_type):
    """
    将整数表示的图像转换为RGB图像

    :param int_image: h*w 的图像np数组，表示整数图像
    :param map_type: 字符串，(fvc/lulc/rsei)
    :return: h*w*3 的图像np数组，表示RGB图像。返回None如果颜色映射中没有找到对应的颜色。
    """
    h, w = int_image.shape
    try:
        color_map = map_type_map[map_type]
        rgb_values = np.array(list(color_map.values()))
        int_values = np.array(list(color_map.keys()))

        # 创建一个与int_image形状相同的数组，用于存储RGB值
        rgb_image = np.zeros((h, w, 3), dtype=np.uint8)

        # 使用广播和np.where进行向量化操作
        for i, int_val in enumerate(int_values):
            rgb_image[int_image == int_val] = rgb_values[i]
        rgb_image = background_2_white(rgb_image, 20)
        return rgb_image
    except KeyError:
        print("颜色映射中没有找到对应的颜色。")
        return None

def background_2_white(image, line_width):
    # Step 1: 创建黑色部分的蒙版
    # 黑色像素定义为所有通道都为 0
    mask = np.all(image == [0, 0, 0], axis=-1).astype(np.uint8)  # 二值蒙版，黑色为1
```

```
    # Step 2: 腐蚀蒙版
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (line_width, line_width))  # 定↙
义腐蚀核
    eroded_mask = cv2.erode(mask, kernel, iterations=1)  # 腐蚀操作

    # Step 3: 替换蒙版区域为白色
    # 将腐蚀后的蒙版区域设为白色
    result_image = image.copy()
    result_image[eroded_mask == 1] = [255, 255, 255]  # 替换为白色
    return result_image

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)



import numpy as np
import matplotlib.colors as mcolors

def get_listed_colormap(map_type):
    """
    根据提供的颜色映射表创建mcolors.ListedColormap

    :param map_type: 字符串，'FVC', 'LULC', 或 'RSEI' 表示所需的颜色映射表
    :return: mcolors.ListedColormap 对象
    """
    # Check if the map_type exists in the map_type_map dictionary
    if map_type in map_type_map:
        # Extract the color map
        color_map = map_type_map[map_type]

        # Extract RGB values and normalize them to [0, 1] for matplotlib
        rgb_values = np.array(list(color_map.values())) / 255.0
        rgb_values[:, [0, 2]] = rgb_values[:, [2, 0]]
        # print(rgb_values.shape)
        # Create and return a ListedColormap
        return mcolors.ListedColormap(rgb_values, name=map_type)
    else:
        print("指定的 map_type 不存在于颜色映射中。")
        return None

def plot_color_mapped_image(int_image, map_type):
    """
    根据整数图像及其颜色映射，在plt上绘制出对应的RGB图像

    :param int_image: h*w 的图像np数组，表示整数图像
    :param map_type: 字符串，(fvc/lulc/rsei)
    """
    rgb_image = img_2d_to_3d_vector(int_image, map_type)
    if rgb_image is not None:
        plt.imshow(rgb_image)
```

```
        plt.axis('off')  # 隐藏坐标轴
        plt.title(f'{map_type} Color Mapped Image')
        plt.show()
    else:
        print("无法绘制图像，因为颜色映射中没有找到对应的颜色。")

# plot_color_mapped_image(predicted_map, "FVC")

# colors = []

# for i in fvc_color_map.values():
#   colors.append((i[2]/255, i[1]/255, i[0]/255))

# print(colors)

# 创建 ListedColormap 对象
# cmap = mcolors.ListedColormap(colors)
cmap = get_listed_colormap(map_type)



import matplotlib.pyplot as plt

def plot_heatmap_square(data):
    """
    绘制一个二维NumPy数组的热力图，每个数据点为正方形。

    Args:
        data: 二维NumPy数组。
    """
    fig, ax = plt.subplots()
    # im = ax.imshow(data, cmap='RdYlGn')  # 选择合适的颜色映射
    im = ax.imshow(data, cmap=cmap)  # 选择合适的颜色映射

    # 隐藏坐标轴
    ax.set_axis_off()

    # 设置纵横比，使单元格为正方形
    ax.set_aspect('equal')

    # 添加颜色条
    fig.colorbar(im, ax=ax)
    plt.tight_layout()  # 调整布局，防止颜色条被裁剪
    plt.show()

# plot_heatmap_square(latest_15_years_data[-5])
# plot_heatmap_square(latest_15_years_data[-4])
# plot_heatmap_square(latest_15_years_data[-3])
# plot_heatmap_square(latest_15_years_data[-2])
plot_heatmap_square(latest_15_years_data[-1])
plot_heatmap_square(predicted_map)

print(latest_15_years_data[-1].min(), latest_15_years_data[-1].max())
unique_values, counts = np.unique(latest_15_years_data[-1], return_counts=True)
print(dict(zip(unique_values, counts)))
```

```
print(predicted_map.min(), predicted_map.max())
unique_values, counts = np.unique(predicted_map, return_counts=True)
print(dict(zip(unique_values, counts)))

import matplotlib.pyplot as plt

def plot_heatmap_square(data):
    """
    绘制一个二维NumPy数组的热力图，每个数据点为正方形。

    Args:
        data: 二维NumPy数组。
    """
    fig, ax = plt.subplots()
    im = ax.imshow(data, cmap='RdYlGn')  # 选择合适的颜色映射
    # im = ax.imshow(data, cmap=cmap)  # 选择合适的颜色映射

    # 隐藏坐标轴
    ax.set_axis_off()

    # 设置纵横比，使单元格为正方形
    ax.set_aspect('equal')

    # 添加颜色条
    fig.colorbar(im, ax=ax)
    plt.tight_layout()  # 调整布局，防止颜色条被裁剪
    plt.show()

# plot_heatmap_square(latest_15_years_data[-5])
# plot_heatmap_square(latest_15_years_data[-4])
# plot_heatmap_square(latest_15_years_data[-3])
# plot_heatmap_square(latest_15_years_data[-2])
plot_heatmap_square(latest_15_years_data[-1])
plot_heatmap_square(predicted_map)

print(latest_15_years_data[-1].min(), latest_15_years_data[-1].max())
unique_values, counts = np.unique(latest_15_years_data[-1], return_counts=True)
print(dict(zip(unique_values, counts)))
print(predicted_map.min(), predicted_map.max())
unique_values, counts = np.unique(predicted_map, return_counts=True)
print(dict(zip(unique_values, counts)))

np.savez_compressed('outputs/RSEI_predicted_map.npz', predicted_map=predicted_map)

"""# graphs"""

# 导入必要的包
import torch
import numpy as np
from tqdm import tqdm
from google.colab.patches import cv2_imshow
import cv2


fvc_color_map = {
```

```
    0: [0, 0, 0],  # 空
    1: [0, 56, 168],  # L
    2: [0, 115, 38],  # H
    3: [82, 142, 249],  # RL
    4: [103, 203, 134],  # RH
    5: [190, 255, 255],  # M
    6: [255, 255, 255],  # 白色像素
}

lulc_color_map = {
    0: [0, 0, 0],  # 空
    1: [115, 255, 255],  # Cropland
    2: [0, 168, 112],  # Forest
    3: [190, 255, 233],  # Grassland
    4: [230, 92, 0],  # Water
    5: [0, 76, 230],  # Impervious
    6: [52, 52, 52],  # Barren
    7: [204, 204, 204],  # Snow/Ice
    8: [255, 255, 255],  # 白色像素
}

rsei_color_map = {
    0: [0, 0, 0],  # 空
    1: [0, 0, 255],  # Poor
    2: [0, 128, 255],  # Fair
    3: [0, 255, 255],  # Moderate
    4: [0, 212, 141],  # Good
    5: [0, 168, 56],  # Excellent
    6: [255, 255, 255],  # 白色像素
}

map_type_map = {
    'FVC': fvc_color_map,
    'LULC': lulc_color_map,
    'RSEI': rsei_color_map,
}

def img_2d_to_3d_vector(int_image, map_type):
    """
    将整数表示的图像转换为RGB图像

    :param int_image: h*w 的图像np数组，表示整数图像
    :param map_type: 字符串，(fvc/lulc/rsei)
    :return: h*w*3 的图像np数组，表示RGB图像。返回None如果颜色映射中没有找到对应的颜色。
    """
    h, w = int_image.shape
    try:
        color_map = map_type_map[map_type]
        rgb_values = np.array(list(color_map.values()))
        int_values = np.array(list(color_map.keys()))

        # 创建一个与int_image形状相同的数组，用于存储RGB值
        rgb_image = np.zeros((h, w, 3), dtype=np.uint8)

        # 使用广播和np.where进行向量化操作
```

```
        for i, int_val in enumerate(int_values):
            rgb_image[int_image == int_val] = rgb_values[i]
        rgb_image = background_2_white(rgb_image, 20)
        return rgb_image
    except KeyError:
        print("颜色映射中没有找到对应的颜色。")
        return None

def background_2_white(image, line_width):
    # Step 1: 创建黑色部分的蒙版
    # 黑色像素定义为所有通道都为 0
    mask = np.all(image == [0, 0, 0], axis=-1).astype(np.uint8)  # 二值蒙版，黑色为1

    # Step 2: 腐蚀蒙版
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (line_width, line_width))  # 定
义腐蚀核
    eroded_mask = cv2.erode(mask, kernel, iterations=1)  # 腐蚀操作

    # Step 3: 替换蒙版区域为白色
    # 将腐蚀后的蒙版区域设为白色
    result_image = image.copy()
    result_image[eroded_mask == 1] = [255, 255, 255]  # 替换为白色
    return result_image

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model_dict = {
    "CNN": PureCNNModel(map_type=map_type),
    "LSTM": PureLSTMModel(map_type=map_type),
    "CNNLSTM": CNN_LSTM_Model(map_type=map_type),
    "Attention": CNN_LSTM_Attention_Model(map_type=map_type),
    "ViT_Trans": ViTTransformer(map_type=map_type),
    "ViT_LSTM": ViTLSTM(map_type=map_type)
}

model_predict_maps = {'CNN': None, 'LSTM': None, 'CNNLSTM': None, 'Attention': None}

# 载入数据并提取最后15年的数据
fvc_data = np.load(f'{map_type}.npz')['arr_0']
mask = np.load('whole_mask.npz')['arr_0']
latest_15_years_data = fvc_data[-15:]  # 提取最后15年的数据，形状为 (15, 4416, 5786)

predicted_map = np.zeros(latest_15_years_data.shape[1:])  # 初始化一个数组来保存预测结果

mask = np.load('whole_mask.npz')['arr_0']

# 设置区域大小
region_size = 10
offset = region_size // 2
height, width = latest_15_years_data.shape[1:]


indices = np.nonzero(mask[offset:height - offset, offset:width - offset])
indices = list(zip(indices[0] + offset, indices[1] + offset))
```

```
for model_type in ['CNN', 'LSTM', 'CNNLSTM', 'Attention']:
  print(model_type)
  # 加载保存的模型
  model = model_dict[model_type].to(device)
  print(f'models/{map_type}/{model_type}.pth')
  model.load_state_dict(torch.load(f'models/{map_type}/{model_type}.pth', weights_only=True))

  model.eval()  # 切换模型到评估模式
  # 批量大小
  batch_size = 1024 * 8  # 可以尝试调整批量大小，根据设备内存情况选择合适的数值

  # 初始化结果数组
  model_predict_maps[model_type] = np.zeros(latest_15_years_data.shape[1:], dtype=np.uint8)

  # 将 indices 划分为批次
  batches = [indices[i:i + batch_size] for i in range(0, len(indices), batch_size)]

  # 遍历每个批次
  for batch in tqdm(batches):
      # 构造批量输入
      input_batch = []
      for i, j in batch:
          input_data = latest_15_years_data[:, i - offset:i + offset + 1, j - offset:j + offset + 1]
          input_batch.append(input_data)

      # 转换为 Tensor 并移动到设备
      input_tensor = torch.tensor(np.array(input_batch), dtype=torch.float32).to(device)  # 形状为 (batch_size, 15, 11, 11)

      # 执行批量预测
      with torch.no_grad():
          outputs = model(input_tensor)
          # 去除白色像素
          outputs = outputs[:, :-1]
          _, predicted_classes = torch.max(outputs, 1)  # 获取预测的类别
          # print(outputs)
          # print(outputs[:-1])

      # 将预测结果写入预测地图
      for (i, j), predicted_class in zip(batch, predicted_classes):
          predicted_map[i, j] = predicted_class.item() + 1  # 提取数值并写入最终结果

  # 画图
  rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
  # cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
  # cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
  cv2_imshow(rgb_image)
  if model_type == 'CNN':
    cv2.imwrite(f'outputs/{map_type}_2023_predicted_map.png', img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
    cv2.imwrite(f'outputs/{map_type}_2022_predicted_map.png', img_2d_to_3d_vector
```

```
(latest_15_years_data[-2], map_type))
    cv2.imwrite(f'outputs/{map_type}_2021_predicted_map.png', img_2d_to_3d_vector↙
(latest_15_years_data[-3], map_type))
    cv2.imwrite(f'outputs/{map_type}_2020_predicted_map.png', img_2d_to_3d_vector↙
(latest_15_years_data[-4], map_type))
  cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)

rgb_image = img_2d_to_3d_vector(predicted_map, map_type)
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-1], map_type))
cv2_imshow(img_2d_to_3d_vector(latest_15_years_data[-2], map_type))
cv2_imshow(rgb_image)
cv2.imwrite(f'outputs/{map_type}_{model_type}_predicted_map.png', rgb_image)
```