Supplementary Material:

MBGCNet: A Novel Fusion of Mini-Batch Graph Convolutional Network and Convolutional Neural Network for PolSAR Image Classification

Introduction

This supplementary document provides a comprehensive explanation of the implementation details of the MBGCNet framework introduced in the main paper, "MBGCNet: A Novel Fusion of Mini-Batch Graph Convolutional Network and Convolutional Neural Network for PolSAR Image Classification." The purpose of this supplement is to facilitate transparency, reproducibility, and ease of understanding for researchers and practitioners interested in replicating or extending our work. Here, we describe the overall architecture, the main components of the codebase, training procedures, data preprocessing, and key implementation choices including the design of the Mini-Batch Graph Convolutional Network (MB-GCN) and its integration with the Convolutional Neural Network (CNN) branch. Detailed comments and explanations are provided to clarify how different modules interact, how the graph structures are constructed dynamically during training, and how the mini-batch training strategy efficiently reduces computational overhead without sacrificing model performance. Additionally, instructions for running experiments and customizing parameters are included to support practical usage. We hope this supplementary material will serve as a valuable resource to aid understanding and foster further research in the field of deep learning for PolSAR image classification.

```
# Environment Setup and Required Libraries
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
from tqdm import tqdm
```

This initial part of the code imports essential libraries and tools used throughout the implementation and evaluation of the model. PyTorch serves as the core deep learning framework, providing modules for building neural network layers and performing numerical computations efficiently on both CPU and GPU. The Dataset and DataLoader utilities facilitate efficient and organized loading of data in mini-batches during training. Additionally, libraries like NumPy and SciPy support numerical data processing, while Matplotlib and Seaborn are used for visualization of results. The tqdm library provides progress bars to monitor the training process, and scikit-learn's metrics modules enable calculation of performance measures such as confusion matrices and classification reports. Together, these imports establish a robust and standard foundation for developing, training, and assessing the MBGCNet framework.

```
# Min-Max Normalization
def min_max_normalize(data):
    data_min = data.min(axis=(0, 2, 3), keepdims=True)
    data_max = data.max(axis=(0, 2, 3), keepdims=True)
    return (data - data_min) / (data_max - data_min + 1e-8)
```

This function implements Min-Max normalization, a common preprocessing step used to scale input data features into the range [0, 1]. The goal is to standardize the data so that each feature has a uniform scale, which helps improve model training stability and convergence. Specifically, the function calculates the minimum and maximum values of the input data along specified dimensions (typically across spatial dimensions for each channel or feature). It then normalizes each data point by subtracting the minimum and dividing by the range (maximum minus minimum). A small epsilon (1e-8) is added to the denominator to avoid potential division-by-zero errors. This type of normalization is especially beneficial for multi-dimensional data such as images or multi-channel features, ensuring that all features contribute proportionally during model training.

```
# Build adjacency matrix with Gaussian similarity
def build_adjacency_matrix(gcn_feats, sigma=1.0):
    num_nodes = gcn_feats.shape[0]
    feats = gcn_feats
    dist = torch.cdist(feats, feats, p=2)
    adj = torch.exp(-dist**2 / (2 * sigma**2))
    degree = torch.sum(adj, dim=1)
    degree_inv_sqrt = torch.pow(degree + 1e-8, -0.5)
    degree_inv_sqrt[torch.isinf(degree_inv_sqrt)] = 0
    D_inv_sqrt = torch.diag(degree_inv_sqrt)
    adj_norm = D_inv_sqrt @ adj @ D_inv_sqrt
    return adj_norm
```

This function constructs a normalized adjacency matrix based on Gaussian similarity, which is essential for capturing the relationships between nodes in the graph convolutional network (GCN) branch.First, it computes the pairwise Euclidean distances between node feature vectors using torch.cdist. These distances are then converted into similarity scores using a Gaussian (RBF) kernel, where the similarity between nodes decreases exponentially with the squared distance, controlled by the parameter sigma. This results in a weighted adjacency matrix representing the strength of connections between nodes. To ensure numerical stability and improve the performance of graph convolutions, the adjacency matrix is symmetrically normalized. This normalization involves computing the degree matrix (sum of edge weights per node), taking its inverse square root, and applying it from both sides to the adjacency matrix. Small epsilon values are added to avoid division by zero, and any infinities resulting from this operation are set to zero. The output is a normalized adjacency matrix that better preserves the graph structure and balances the influence of nodes during message passing in the GCN.

```
# Dataset class with multi-patch sampling
class PolSARPatchDataset(Dataset):
    def __init__(self, patches, labels, num_patches_per_sample=5, patch_size=32):
        self.patches = patches
        self.labels = labels
        self.num_patches_per_sample = num_patches_per_sample
        self.patch_size = patch_size
        self.num_samples = patches.shape[0]
        self.channels = patches.shape[1]

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        cnn_input = self.patches[idx]
        gcn_indices = np.random.choice(len(self.patches), self.num_patches_per_sample,
replace=False)
        gcn_patches = [self.patches[i] for i in gcn_indices]
        gcn_feats = np.stack([patch.reshape(self.channels, -1).T for patch in gcn_patches],
axis=0)
        label = self.labels[idx]
        return {
            'cnn_input': torch.from_numpy(cnn_input).float(),
            'gcn_patches': torch.from_numpy(np.array(gcn_patches)).float(),
            'label': torch.tensor(label, dtype=torch.long)
        }
```

This class defines a custom PyTorch Dataset tailored for the MBGCNet framework, enabling multi-patch sampling from PolSAR image data. Each dataset sample consists of spatial patches and their corresponding labels.

The constructor (__init__) initializes the dataset with input patches, labels, the number of patches to sample per graph convolution input, and the patch size. It also stores the total number of samples and the number of data channels.

The __len__ method returns the total number of samples in the dataset, allowing PyTorch to iterate through the data properly.

The __getitem__ method retrieves a single sample at the given index. It prepares two main inputs for the model:

- cnn_input: the original patch at the specified index, used for the CNN branch.
- gcn_patches: a randomly selected set of patches (without replacement) used to build the local graph for the GCN branch. These patches are reshaped and stacked into feature vectors suitable for graph processing.

The corresponding label is also returned as a tensor. This design supports the hybrid architecture by supplying both localized spatial information (for CNN) and contextual graph-based features (for GCN) from multiple patches within the dataset.

```
# Collate function
def collate_fn(batch):
    batch_size = len(batch)
    cnn_inputs = torch.stack([x['cnn_input'] for x in batch])
    labels = torch.stack([x['label'] for x in batch])
    gcn_patches = [x['gcn_patches'] for x in batch]
    N = gcn_patches[0].shape[0]
    C = gcn_patches[0].shape[1]
    M = 32
    gcn_feats = torch.cat([patches.reshape(N, C, M * M).permute(0, 2, 1).reshape(N * M * M,
C) for patches in gcn_patches], dim=0)
    adj_norm = build_adjacency_matrix(gcn_feats)
    return cnn_inputs, gcn_feats, adj_norm, labels
```

This custom collate function is designed to efficiently batch and prepare data samples for the MBGCNet training pipeline, particularly handling the hybrid inputs for both the CNN and GCN branches. Given a batch of samples, the function first stacks the CNN input patches and their corresponding labels into tensors for straightforward batch processing. For the GCN branch, it processes the multiple sampled patches from each batch element by reshaping and concatenating them into a unified feature tensor suitable for graph construction. Specifically, each set of GCN patches is reshaped to flatten the spatial dimensions, then permuted to organize the features correctly before concatenation. This results in a large tensor containing node features from all patches across the batch. Next, the function calls the build_adjacency_matrix method to compute a normalized adjacency matrix based on the combined node features, capturing the graph connectivity needed for the GCN convolution operations. Finally, the function returns the CNN batch inputs, the flattened GCN node features, the normalized adjacency matrix, and the batch labels, ready to be fed into the MBGCNet model for joint local-global feature learning.

```
# GCN Layer for MB-GCN-MPS
class MBGCNLayer(nn.Module):
    def __init__(self, in_features, out_features, patch_size=32, num_patches=5):
        super(MBGCNLayer, self).__init__()
        self.patch_size = patch_size
        self.num_patches = num_patches
        self.conv1x1 = nn.Conv1d(in_features, out_features, kernel_size=1, bias=True)
        nn.init.kaiming_normal_(self.conv1x1.weight)  # He Initialization
        self.layer_norm = nn.LayerNorm([patch_size * patch_size, out_features])
        self.dropout = nn.Dropout(0.2)

    def forward(self, x, adj):
        # x: [B * N * M^2, F_in], adj: [B * N * M^2, B * N * M^2]
        h = torch.mm(adj, x)  # [B * N * M^2, F_in]
        h = h.reshape(-1, self.num_patches * self.patch_size * self.patch_size, x.size(1))
# [B, N * M^2, F_in]
        h = h.permute(0, 2, 1)  # [B, F_in, N * M^2]
        h = self.conv1x1(h)  # [B, F_out, N * M^2]
        h = h.permute(0, 2, 1)  # [B, N * M^2, F_out]
        h = h.reshape(-1, self.patch_size * self.patch_size, h.size(2))  # [B * N, M^2,
F_out]
        h = self.layer_norm(h)
        h = self.dropout(h)
        return F.relu(h)
```

This class implements a single graph convolutional layer specifically designed for the Mini-Batch Graph Convolutional Network (MB-GCN) used in the MBGCNet framework. The layer accepts input node features and a normalized adjacency matrix representing graph connectivity. It performs graph convolution by aggregating features from neighboring nodes via matrix multiplication of the adjacency matrix and the input features. Key components and workflow include:

- 1x1 Convolution (conv1x1): After the graph aggregation step, a pointwise convolution (1D convolution with kernel size 1) is applied along the feature dimension. This operation linearly transforms the aggregated features, enabling flexible feature extraction while maintaining spatial structure.
- Reshaping and Permutation: To apply the convolution properly, the data is reshaped and permuted to organize nodes and features in a format compatible with the convolution operation. The input tensor dimensions transition through multiple forms to handle batches (B), number of patches (N), patch size (M), and feature channels (F).
- Layer Normalization: Applied to stabilize training by normalizing feature activations across nodes in each patch. This enhances convergence and generalization.
- Dropout: Added as regularization to prevent overfitting by randomly zeroing some of the layer outputs during training.
- Activation: Finally, a ReLU activation function introduces non-linearity to the output, enabling the model to learn complex feature mappings.

Together, these components enable the MBGCNLayer to efficiently learn meaningful representations from local graph-structured polarimetric features extracted from multiple spatial patches, while maintaining scalability through mini-batch processing.

```
# GCN Layer for MB-GCN-MPS
class MBGCNLayer(nn.Module):
    def __init__(self, in_features, out_features, patch_size=32, num_patches=5):
        super(MBGCNLayer, self).__init__()
        self.patch_size = patch_size
        self.num_patches = num_patches
        self.conv1x1 = nn.Conv1d(in_features, out_features, kernel_size=1, bias=True)
        nn.init.kaiming_normal_(self.conv1x1.weight)  # He Initialization
        self.layer_norm = nn.LayerNorm([patch_size * patch_size, out_features])
        self.dropout = nn.Dropout(0.2)

    def forward(self, x, adj):
        # x: [B * N * M^2, F_in], adj: [B * N * M^2, B * N * M^2]
        h = torch.mm(adj, x)  # [B * N * M^2, F_in]
        h = h.reshape(-1, self.num_patches * self.patch_size * self.patch_size, x.size(1))
# [B, N * M^2, F_in]
        h = h.permute(0, 2, 1)  # [B, F_in, N * M^2]
        h = self.conv1x1(h)  # [B, F_out, N * M^2]
        h = h.permute(0, 2, 1)  # [B, N * M^2, F_out]
        h = h.reshape(-1, self.patch_size * self.patch_size, h.size(2))  # [B * N, M^2,
F_out]
        h = self.layer_norm(h)
        h = self.dropout(h)
        return F.relu(h)
```

The MBGCNet class defines the overall Mini-Batch Graph Convolutional Network with Multi-Patch Sampling (MB-GCN-MPS) model, designed for PolSAR image classification by synergistically combining spatial convolutional and graph-based feature learning.

Architecture overview:

- CNN Branch: This branch extracts local spatial features directly from input patches using a sequence of convolutional layers. It employs three convolutional blocks with progressively increasing channel dimensions (32, 64, 128), each followed by batch normalization, ReLU activation, and downsampling (max pooling or stride). The final spatial features are aggregated via an adaptive average pooling layer into a fixed-length vector. He initialization is applied to all convolutional layers to ensure stable and efficient training.
- GCN Branch: To capture the structural relationships between spatial features across multiple patches, the model includes a two-layer graph convolutional network (GCN) implemented by MBGCNLayer modules. Input features for the GCN come from reshaped multi-patch data, and adjacency matrices encode feature similarity using Gaussian kernels. The GCN outputs contextualized node embeddings per patch. The model then extracts the feature corresponding to the patch center and aggregates across patches by averaging to form a global graph representation.
- Feature Fusion and Classification: The spatial CNN feature vector and the aggregated GCN graph embedding are concatenated, dropout regularization is applied, and the fused feature vector is passed through a fully connected classifier. The classifier consists of two linear layers with ReLU activation and dropout, culminating in output logits corresponding to the target classes. The linear layers are also initialized with He initialization to promote effective learning.

Forward pass summary:

- The CNN branch processes input image patches to produce spatial features.
- The GCN branch processes multi-patch graph-structured data to learn relational features.
- Features from both branches are fused and jointly used for final classification.

This hybrid architecture leverages the complementary strengths of convolutional feature extraction and graph-based relational learning to improve classification accuracy in complex PolSAR image scenarios.

```
# Training and Evaluation Functions
def train_one_epoch(model, dataloader, optimizer, criterion, device):
    model.train()
    total_loss = 0
    correct = 0
    total = 0
    for cnn_input, gcn_feats, adj, labels in tqdm(dataloader, leave=False):
        cnn_input, gcn_feats, adj, labels = cnn_input.to(device), gcn_feats.to(device),
adj.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(cnn_input, gcn_feats, adj)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * labels.size(0)
        pred = outputs.argmax(dim=1)
        correct += (pred == labels).sum().item()
        total += labels.size(0)
    return total_loss / total, correct / total

def evaluate(model, dataloader, device):
    model.eval()
    y_true, y_pred = [], []
    with torch.no_grad():
        for cnn_input, gcn_feats, adj, labels in dataloader:
            cnn_input, gcn_feats, adj = cnn_input.to(device), gcn_feats.to(device),
adj.to(device)
            outputs = model(cnn_input, gcn_feats, adj)
            pred = outputs.argmax(dim=1).cpu().numpy()
            y_pred.extend(pred)
            y_true.extend(labels.cpu().numpy())
    return np.array(y_true), np.array(y_pred)
```

This section describes the training and evaluation procedures implemented for the MBGCNet model.

Training function (train_one_epoch): The model is trained for one epoch over the provided dataset. The function iterates through the dataloader batches, each containing CNN input patches, GCN features, adjacency matrices, and labels. For each batch:

- The data is transferred to the specified computing device (CPU/GPU).
- The model's gradients are reset before the forward pass.
- The model outputs predictions by processing both CNN and GCN inputs.
- The cross-entropy loss between predictions and ground truth labels is computed.
- Backpropagation is performed by calling loss.backward().
- The optimizer updates the model parameters accordingly.
- The cumulative loss and the number of correct predictions are tracked to compute the average loss and accuracy over the epoch.

This function returns the mean loss and accuracy, providing key metrics for monitoring training progress.

Evaluation function (evaluate): During evaluation, the model is set to inference mode, disabling dropout and gradient computations for efficiency. The function iterates over the evaluation dataset, computing model predictions on each batch. The predicted and true labels are collected for the entire dataset and returned as numpy arrays, enabling further performance analysis such

as confusion matrices and classification reports. Together, these functions support an effective supervised training loop and accurate performance assessment for the MBGCNet framework.

```
# Data Splitting Function
def split_by_class_percent(patches, labels, percent=0.01, seed=42):
    np.random.seed(seed)
    unique_classes = np.unique(labels)
    train_idx, test_idx = [], []
    for cls in unique_classes:
        idxs = np.where(labels == cls)[0]
        np.random.shuffle(idxs)
        n_train = max(1, int(len(idxs) * percent))
        train_idx.extend(idxs[:n_train])
        test_idx.extend(idxs[n_train:])
    return np.array(train_idx), np.array(test_idx)
```

This function performs a stratified split of the dataset into training and testing subsets based on a specified percentage of samples per class. Given the input patches and their corresponding labels, it divides the data while maintaining class distribution proportionally.

- The random seed is set for reproducibility.
- For each unique class in the dataset, the indices of samples belonging to that class are identified and shuffled randomly.
- A number of training samples for each class is calculated as the maximum between one and the specified percentage of that class's total samples, ensuring at least one training example per class.
- The first portion of the shuffled indices is assigned to the training set, while the remainder forms the testing set.
- Finally, the function returns two arrays containing the indices of training and testing samples, respectively.

This stratified splitting approach ensures balanced class representation in both subsets, which is crucial for training and evaluating classification models reliably.

```
# Patch Extraction Function
def extract_patches(img, lbl, patch_size=32, stride=16):
    H, W, C = img.shape
    patches, patch_labels = [], []
    for r in range(0, H - patch_size + 1, stride):
        for c in range(0, W - patch_size + 1, stride):
            patch = img[r:r + patch_size, c:c + patch_size, :]
            patch_label = lbl[r:r + patch_size, c:c + patch_size]
            label = np.bincount(patch_label.flatten()).argmax()
            patches.append(np.transpose(patch, (2, 0, 1)))
            patch_labels.append(label)
    return np.array(patches), np.array(patch_labels)
```

This function extracts overlapping spatial patches from a hyperspectral image along with their corresponding labels.

- The input includes a 3D image tensor (img) with dimensions Height × Width × Channels, and a 2D label map (lbl) with spatial dimensions matching the image.

- The function slides a fixed-size window (patch) over the image with a specified stride, extracting patches of size patch_size × patch_size.
- For each patch location, the corresponding label patch is extracted from the label map. The final label assigned to the patch is determined by the majority class within the patch, computed via np.bincount on the flattened label patch.
- Each extracted image patch is transposed to a channel-first format (Channels × Height × Width), aligning with common deep learning input conventions.
- The function collects all extracted patches and their labels into arrays and returns them for downstream processing such as training or evaluation.

This patch extraction method enables capturing local spatial context while preserving class information, which is crucial for effective learning in spatial-spectral classification tasks.

```
# Main Script
if __name__ == "__main__":

    # Load PolSAR data and GTM
    data = scipy.io.loadmat('/PolSAR.mat')
    gtm = scipy.io.loadmat('/GTM.mat')
    PolSAR = data['PolSAR']  # PolSAR image data
    gt = gtm['GTM']  # GTM

    # Display one band of PolSAR data (e.g., Band 1)
    plt.figure(figsize=(8, 6))
    plt.imshow(PolSAR[:, :, 0], cmap='gray')
    plt.colorbar()
    plt.title("PolSAR Band 1")
    plt.show()
    # Display the GTM
    plt.figure(figsize=(8, 6))
    plt.imshow(gt, cmap='jet')  # 'jet' provides better visualization for categorical data
    plt.colorbar()  # Add a color legend
    plt.title("Ground Truth Map (GTM)")
    plt.show()
    image = data['PolSAR']
    labels = gtm['GTM']

    # Extract and normalize patches
    patches, patch_labels = extract_patches(image, labels, patch_size=32, stride=16)
    patch_labels = patch_labels - 1
    patches = min_max_normalize(patches)

    # Split data
    train_idx, test_idx = split_by_class_percent(patches, patch_labels, percent=0.01)
    train_patches, train_labels = patches[train_idx], patch_labels[train_idx]
    test_patches, test_labels = patches[test_idx], patch_labels[test_idx]

    # Create datasets and dataloaders
    train_dataset = PolSARPatchDataset(train_patches, train_labels,
num_patches_per_sample=5)
    test_dataset = PolSARPatchDataset(test_patches, test_labels, num_patches_per_sample=5)
    train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True,
collate_fn=collate_fn)
```

```
    test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False,
collate_fn=collate_fn)

    # Model setup
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    num_classes = len(np.unique(patch_labels))
    input_channels = patches.shape[1]
    model = MBGCNet(in_channels=input_channels, num_classes=num_classes).to(device)
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-4)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=100)
    criterion = nn.CrossEntropyLoss()

    # Training loop
    for epoch in range(100):
        train_loss, train_acc = train_one_epoch(model, train_loader, optimizer, criterion,
device)
        scheduler.step()
        print(f"Epoch {epoch+1} Train loss: {train_loss:.4f} Acc: {train_acc*100:.4f}")

    y_true, y_pred = evaluate(model, test_loader, device)
    print(classification_report(y_true, y_pred))
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.savefig('confusion_matrix.png')
```

This script implements a complete pipeline for supervised classification of Polarimetric Synthetic Aperture Radar (PolSAR) data using a deep learning framework. The process begins by loading the PolSAR image data along with its corresponding Ground Truth Map (GTM) from MATLAB .mat files. The PolSAR dataset consists of multi-channel image data representing different polarization bands, while the GTM provides pixel-wise class labels for supervised training. To facilitate intuitive understanding, the script visualizes one selected spectral band of the PolSAR data using a grayscale colormap, illustrating the underlying radar backscatter intensities. It also displays the GTM with a jet colormap, which enhances the visibility of distinct classes by assigning unique colors to each category. These visualizations provide insight into the spatial distribution and class boundaries in the dataset. Next, the core data preparation stage involves extracting smaller spatial patches from the large PolSAR image using a sliding window approach with specified patch size and stride. Each extracted patch is labeled by the majority class within its corresponding region in the GTM, ensuring that the patch's label represents the dominant land cover or target type. Subsequently, the extracted patches undergo min-max normalization to scale their pixel intensity values into a standardized range, improving model convergence and performance. For training and evaluation, the dataset is split on a per-class basis by randomly selecting a small percentage (1%) of samples from each class for training, while the remainder are reserved for testing. This balanced split strategy guarantees that all classes are represented in the training set, which is especially important in scenarios with class imbalance. The training and testing samples are then wrapped into custom dataset objects, which support the generation of multiple patches per sample to capture local spatial context. These datasets are loaded into PyTorch DataLoaders to efficiently batch, shuffle, and collate samples during model training and evaluation. The model architecture, instantiated here as MBGCNet, is initialized with the appropriate number of input channels and output classes, then moved to the available compute device (GPU if available). The training

setup includes the AdamW optimizer for adaptive gradient-based updates, a cosine annealing learning rate scheduler to progressively reduce the learning rate for smoother convergence, and cross-entropy loss to measure classification error. The training proceeds for 100 epochs, where in each epoch the model learns from the training dataset, optimizing its parameters to minimize classification loss and maximize accuracy. The learning rate is adjusted after every epoch according to the cosine annealing schedule. After training completes, the model is evaluated on the test dataset to obtain predicted labels. The script then generates a comprehensive classification report detailing precision, recall, F1-score, and overall accuracy per class, providing quantitative insights into model performance. Finally, a confusion matrix is computed and visualized using a heatmap, illustrating the distribution of true versus predicted labels and highlighting any common misclassifications. The confusion matrix figure is saved for documentation and further analysis. In summary, this script encapsulates the entire supervised learning workflow for PolSAR image classification—from data loading and preprocessing, through model training and evaluation, to detailed performance visualization—providing a robust foundation for research and application in radar image analysis.

How to Run the Code, MBGCNet-MPS: PolSAR Image Classification

This repository contains the implementation of **MBGCNet-MPS**, a hybrid deep learning framework combining CNN and Mini-Batch Graph Convolutional Network (MB-GCN) for PolSAR (Polarimetric Synthetic Aperture Radar) image classification.

https://github.com/mohsendarvishnezhad/MBGCNet

1. Prerequisites

- Python 3.7 or higher

- Required Python packages:

  - `torch`

  - `numpy`

  - `scipy`

  - `matplotlib`

  - `scikit-learn`

  - `seaborn`

  - `tqdm`

You can install all required packages via pip:

pip install torch numpy scipy matplotlib scikit-learn seaborn tqdm

2. Data Preparation

Prepare your PolSAR data as spatial patches saved in a `.mat` file. The `.mat` file should contain two variables:

- `patches`: a NumPy array of shape `[num_samples, num_channels, patch_size, patch_size]`

- `labels`: a 1D array containing class labels for each sample

Make sure your data is formatted accordingly before running the code.

3. Configure Data Path

In the main Python script (`MBGCNet_MPS.py`), locate the section:

```python

if __name__ == '__main__':

    data_path = 'path/to/your/polsar_patches.mat'

```

Replace `'path/to/your/polsar_patches.mat'` with the actual path to your `.mat` data file.

4. Run the Script

Execute the script in your terminal or command prompt:

python MBGCNet_MPS.py

The training process will start, and after each epoch, you will see the training progress and accuracy printed in the console.

5. GPU Usage (Optional)

If a CUDA-compatible NVIDIA GPU is available and PyTorch detects it, the code will automatically utilize the GPU for faster training.

Notes

- To train the model on your own data, adjust the data path and ensure the data format matches the expected input.

- You can modify training parameters such as batch size, number of epochs, and learning rate by editing the script.

- For any issues or questions, feel free to contact or open an issue in the repository.