7. Training Script (train.py)

This script orchestrates the entire learning process. It loads data, feeds it to the model, calculates errors, and updates the model to improve.

1. Imports and Setup

```
import os
import argparse
# ... imports ...
from torch.utils.tensorboard import SummaryWriter
```

- `argparse`: Handles command-line arguments (e.g., setting batch size or learning rate from the terminal).
- `SummaryWriter`: Enables **TensorBoard** logging, allowing you to visualize loss curves and metrics in real-time.

2. Argument Parsing (`get_args`)

```
def get_args():
    parser = argparse.ArgumentParser(...)
    parser.add_argument('--data_root', ..., required=True)
    # ...
    return parser.parse_args()
```

- **Purpose:** Defines the knobs and dials you can turn without changing the code.

- **Key Args:**
  - `--dataset_name`: Switches between SECOND and Landsat logic automatically.
  - `--data_root`: Where your images are stored.
  - `--batch_size`: How many images the GPU processes at once.

3. The Training Loop (`train_one_epoch`)

```
def train_one_epoch(model, loader, criterion, optimizer, device, epoch, writer):
    model.train() # Switch to training mode (enables Dropout/BatchNorm)
```

- `model.train()`: Tells PyTorch "We are learning now." This is crucial because layers like Dropout behave differently during training vs. testing.

```
    for i, batch in enumerate(pbar):
        img_A = batch['img_A'].to(device)
        # ... move all data to GPU ...
```

- `.to(device)`: Moves the images and labels from the CPU (RAM) to the GPU (VRAM) for fast calculation.

```
        # Forward Pass
        outputs = model(img_A, img_B)
```

- `outputs` : The model makes its best guess. This returns the dictionary containing semantic maps, change masks, and flow data.

```
        # Loss Calculation
        loss_dict = criterion(outputs, targets)
        loss = loss_dict["total"]
```

- `criterion` : Calls `LambdaSCDLoss` . It compares the guess (`outputs`) against the reality (`targets`) to calculate a score of how "wrong" the model is.

```
        # Backward Pass (The Learning)
        optimizer.zero_grad()   # Clear old gradients
        loss.backward()         # Calculate new gradients (Backpropagation)
        optimizer.step()        # Update weights
```

- `optimizer.zero_grad()` : If you don't do this, gradients accumulate, and the updates become huge and wrong.
- `loss.backward()` : The "Chain Rule" of calculus runs backward through the network to find out which parameter caused the error.
- `optimizer.step()` : Adjusts the weights slightly in the opposite direction of the error.

```
        # Logging
        if i % 10 == 0:
            writer.add_scalar('Train/Step_Loss', loss.item(), global_step)
```

- **Purpose:** Every 10 steps, it sends the current loss to TensorBoard so you can see if the training is crashing or working.

4. Validation Loop (`validate`)

```
def validate(model, loader, criterion, metrics, device, epoch, writer):
    model.eval()       # Switch to eval mode
    metrics.reset()    # Clear previous scores

    with torch.no_grad(): # Disable gradient calculation
```

- `model.eval()`: Turns off Dropout.
- `torch.no_grad()`: Tells PyTorch not to store the history of operations. This saves massive amounts of memory and speeds up validation since we aren't training here.

```
        # Update Metrics
        pred_sem1 = torch.argmax(outputs['sem1'], dim=1)
        metrics.update(pred_sem1, label_A)
```

- `torch.argmax`: The model outputs probabilities (e.g., `[0.1, 0.8, 0.1]`). Argmax picks the highest one (Index 1).
- `metrics.update`: Adds these predictions to a Confusion Matrix to calculate IoU and F1-score later.

5. Main Execution (`main`)

```
def main():
    # ... setup device and logging ...

    # Initialize Datasets
    train_ds = SCDDataset(..., mode='train', random_flip=True, ...)
    val_ds = SCDDataset(..., mode='val', random_flip=False, ...)
```

- **Augmentation:** Note that `random_flip=True` is only for training. We never flip validation images because we want a consistent benchmark.

```
    # Build Model
    model = LambdaSCD(num_classes=num_classes).to(device)

    # Optimization Setup
    optimizer = optim.AdamW(model.parameters(), lr=args.lr, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.CosineAnnealingLR(...)
```

- `AdamW`: A modern optimizer that handles weight decay (regularization) better than standard Adam.
- `CosineAnnealingLR`: Slowly lowers the learning rate over time. It starts high to learn fast, then gets small to fine-tune the details.

```
    # The Loop
    for epoch in range(1, args.epochs + 1):
        train_loss = train_one_epoch(...)
        val_loss, val_results = validate(...)

        # Save Best Model
        if score > best_score:
```

```
            torch.save(model.state_dict(), 'best_model.pth')
```