

6. Dataset Loader (dataset.py)

This file is the bridge between your hard drive and the neural network. It handles loading images, converting labels, and applying the normalization logic we defined earlier.

1. Dataset Configurations

```
# --- SECOND Dataset Config ---
ST_NUM_CLASSES = 7
ST_COLORMAP = [[255, 255, 255], [0, 0, 255], ...]

# --- LandsatSCD Dataset Config ---
LANDSAT_NUM_CLASSES = 5
LANDSAT_COLORMAP = [[255, 255, 255], [0, 155, 0], ...]
```

- **Purpose:** Defines the rules for each dataset.
- **Colormaps:** Semantic segmentation labels are often stored as standard RGB images (e.g., Water is Blue, Ground is Grey). To train a network, we must convert these colors into integer IDs (Water = 1, Ground = 2).

2. Fast Color Conversion (`build_colormap_lookup`)

```
def build_colormap_lookup(colormap):
    lookup = np.zeros(256 ** 3, dtype=np.uint8)
    for i, cm in enumerate(colormap):
        idx = (cm[0] * 256 + cm[1]) * 256 + cm[2]
        lookup[idx] = i
```

```
    return lookup
```

- **The Problem:** Scanning every pixel in a 1024x1024 image to check `if pixel == [0, 0, 255]` is very slow.
- **The Solution:** We create a "Lookup Table."
 - We treat the RGB color as a base-256 number: $R*256^2 + G*256 + B$.
 - We create a massive array (size $256^3 \approx 16$ million).
 - We place the Class ID at the specific index corresponding to that color.
- **Result:** Converting an entire image becomes an instant array lookup `IndexMap = lookup[Image]`, which is hundreds of times faster than loops.

3. The SCDDataset Class

Initialization (`__init__`)

```
def __init__(self, root, mode, dataset_name='SECOND', ...):  
    # Select Config  
    if 'second' in dataset_name.lower():  
        self.lookup_table = st_colormap2label  
    # ...
```

- **Logic:** It automatically selects the correct Colormap Lookup Table based on the dataset name.
- **Validation:** It checks `_get_valid_file_list` to ensure that for every sample, all 5 required files exist (Image A, Image B, Label A, Label B, Change Mask).

Synchronization (`_sync_transform`)

```
def _sync_transform(self, img_A, img_B, sem1, sem2, bcd):
    if random.random() > 0.5:
        # Flip ALL images and labels together
        img_A = TF.hflip(img_A)
        # ...
        bcd = TF.hflip(bcd)
```

- **Crucial Concept:** In Change Detection, spatial alignment is everything. If you flip Image A to augment data, you **must** flip Image B and all labels in the exact same way. If you flip A but not B, the ground truth becomes wrong (the "change" locations move).

Loading Data (`__getitem__`)

```
def __getitem__(self, idx):
    # 1. Load Images
    img_A = io.imread(path_A)

    # 2. Process Labels
    label_sem1 = Color2Index(label_sem1, self.lookup_table, ...)

    # 3. Process BCD (Binary Change Detection)
    label_bcd = (label_bcd > 0).astype(np.int64)
```

- `Color2Index` : Converts the RGB semantic map into a Class ID map (0, 1, 2...) using the lookup table.
- **BCD Processing:** Ensures the change mask is strictly binary (0 or 1).

```
# 4. Normalize
img_A = norm_utils.normalize_image(img_A, 'A', self.dataset_name)
img_B = norm_utils.normalize_image(img_B, 'B', self.dataset_name)
```

- **Integration:** Calls the `normalization_utils.py` logic we discussed earlier. It passes the dataset name so the correct Mean/Std are used.

```
# 5. Random Swap (Augmentation)
if self.mode == 'train' and self.random_swap:
    if random.random() > 0.5:
        t_img_A, t_img_B = t_img_B, t_img_A
        t_sem1, t_sem2 = t_sem2, t_sem1
```

- **Purpose:** Teaches the model that change is bidirectional. "Grass to Building" (Construction) is just as important as "Building to Grass" (Demolition). By swapping T1 and T2, we double the variety of temporal