Guide: Preparing Data and Running the Thin-Section Mineral Segmentation Workflow

Critical Requirement

Optical input images and corresponding labels must be **aligned pixel-by-pixel**, and labels must be ready for training, either as class IDs or as colors consistently mapped to class IDs.

1 What This Workflow Does

This workflow trains a deep-learning model to generate mineral maps from optical thin-section images. The model learns a mapping between:

- **Input**: Optical microscopy images (single-angle or multi-angle polarized light)
- **Output**: Mineral label masks (ground truth)

Once trained, the model can predict mineral distributions for new thin-section images or for extracted image patches.

2 Hardware Recommendation for Training

Model training is computationally intensive, particularly when using high-resolution patches, multiple optical angles, and deep encoder backbones. For efficient training, the use of a high-end GPU is strongly recommended.

- GPUs with at least **16–24 GB of VRAM** are recommended for multi-angle training.
- Systems with limited GPU memory may require reducing batch size, patch size, or the number of optical angles.
- CPU-only training is supported just for debugging or small-scale tests.

3 What You Need Before Running Anything

Before running any code, ensure that the following requirements are met:

- Python version 3.9 or newer
- All required Python packages installed (see `requirements.txt`)
- A dataset organized by sample
- At least **one aligned optical angle per sample** (minimum requirement)
- A defined list of mineral classes to be predicted
- Labels that are either already aligned or can be generated using the provided scripts

Important: If alignment or label generation is incorrect, model results will be meaningless, even if the code executes without errors.

4 Dataset Layout on Disk

The dataset must follow the structure shown below:

```
data/
  sample_name_1/
    image/   # optical images (*.tif)
    label/   # mineral label masks (*.tif)
  sample_name_2/
    image/
    label/
```

Notes:

- The folder name `data` must match the value specified for `data_root` in `config.yaml`.
- Sample folder names must exactly match those listed under `samples` in the `data` section of `config.yaml`. Names are case-sensitive.

Each sample folder represents one thin section (or one physical specimen). The `image/` folder contains optical images (PPL, XPL, or multiple angles), while the `label/` folder contains a single-channel label mask used for training.

5 Data Preparation: Conceptual Explanation

The model learns pixel-level correspondences between optical images and mineral labels. Therefore:

- Each pixel in the label must correspond to the same physical location in the optical image.
- Labels must be aligned in scale, orientation, and spatial resolution.

5.1 Alignment Requirement (Very Important)

If the data are not already aligned, the following scripts must be used during data preparation:

- `input_alignments.py`: aligns optical inputs and reference maps
- `labelgeneration.py`: generates training-ready label masks

Minimum requirement: At least one optical angle with ground truth labels must be aligned. This aligned angle can then be used to align additional angles or to generate the final label mask.

6 Label Generation and Color Mapping

In many cases, labels originate from sources such as μXRF or mineral maps that use colors or categorical values. These labels must be converted into a **single-channel class-ID mask** prior to training.

This conversion is handled by `labelgeneration.py`.

Required Edits in `labelgeneration.py`

Users must:

- Define input paths, output visualization paths, and output label paths
- Edit `class_names` and corresponding `hex_colors` (or values) to match the label data
- Ensure that each mineral class maps to a unique integer class ID

Incorrect color or class mapping will result in mislabeled training data and invalid model predictions.

7 Editing `config.yaml`

All workflow settings are controlled through a single file, `config.yaml`. Most users do not need to modify any Python source code.

Section	Purpose	Options
project	Experiment metadata	–
data	Dataset and loading	stack, separate
split	Data splitting strategy	fraction, sample_holdout, fixed_manifest
model	Architecture and classes	UNet, UNet++, DeepLabV3+
train	Training control	–
eval	Metrics and visuals	–
inference	Prediction settings	–

More details are provided in the repository README.

8 Running the Code

Step A: Install Dependencies

```
pip install -r requirements.txt
```

Step B: Run a Quick Smoke Test

```
python traintest.py --config config.yaml
```

Step C: Train the Model

```
python -m src.train --config config.yaml
```

Step D: Evaluate the Model

```
python -m src.evaluate --config config.yaml --checkpoint models/best.
    ckpt
```

Step E: Run Inference

```
python inference.py --config config.yaml
```

9 Patch Naming for Inference

```
patch_0001_PPL.tif
patch_0001_XPL40.tif
patch_0001_XPL50.tif
patch_0001_XPL60.tif
...
```

10 Troubleshooting Checklist

- **Files not found**: Verify dataset paths and sample names
- **Incorrect results**: Verify alignment and label color mapping
- **Out-of-memory**: Reduce batch size or workers

Best practice: Maintain a small debug dataset and validate the full pipeline before scaling up.