In this project, we tried to implement a pipeline to detect illegal mining operations in the Amazon using satellite imagery and deep learning. Our approach combined two main ideas:

1. **Semantic Segmentation (SegFormer)**: A pre-trained open source model model takes high overview photos, evaluates which zones look suspicious and send those to a model we trained ourselves specially for detecting mines. The reason we don't just create a grid of the xwhole amazon and feed it into our model is because it takes too much compute. However it is important to note that we do realize that this is indeed adding another possible point of failure.
2. **Binary Classification**: We fine-tuned a CNN classifier to classify whether suspicious regions actually contain mining operations.

Architecture

Our pipeline was built in six independant steps:

Step	Name	Purpose
1	Collect Mines	Gather positive samples (known mining sites) from local Landsat imagery and CSV coordinates
2	Collect Forest	Generate negative samples (forest) from protected areas via Sentinel-2
3	Build Dataset	Create train/validation splits with augmentation and style matching
4	Train Model	Train a CNN classifier to distinguish mining from forest
5	Detect (Two-Stage)	Apply segmentation + classification to scan large geographic areas
6	Validate	Evaluate model performance against known ground-truth coordinates

Imports

The next cells are the imports that can be broken down into multiple categories:

Standard Library: `csv`, `pathlib`, `typing`, `json`, `shutil`, `time`, `random`, `gc` for file In out processing, type hints, and memory management.

Image Processing: `PIL` (Pillow) for image loading/saving, `cv2` (OpenCV) for our operations operations in the segmentation post-processing.

Numerical Computing: `numpy` for array operations on image data.

Satellite Data Access: `planetary_computer` and `pystac_client` to be able to use Microsoft Planetary Computer's Sentinel-2 archive (for the satellite imagery retrieval);

`rasterio` for reading geospatial raster data (special kind of retrieved dta returned by MPC).

Deep Learning: `torch` and `torchvision` for the CNN classifier; `transformers` for the SegFormer segmentation model.

The satellite libraries are needed for Steps 1, 2, 5, and 6. The deep learning libraries are used in Steps 4 and 5.

In [ ]:
```
import csv
from pathlib import Path
from typing import Dict, List, Optional, Tuple
import json
import shutil

from PIL import Image, ImageDraw
import time
import planetary_computer as pc
from pystac_client import Client
import rasterio
from rasterio.warp import transform_bounds
import random
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, random_split
from torchvision import transforms, models
import numpy as np
import cv2
import gc

from transformers import SegformerForSemanticSegmentation, SegformerImagePro
```

Configuration Parameters

This cell defines all configurable parameters for the pipeline. It is contained in the config.py file in the implementation.

Data Paths

- `LOCAL_MINE_DIR` : Directory containing existing Landsat imagery of known mining sites (positive samples).
- `MINE_COORDINATES_CSV` : CSV file with latitude/longitude coordinates of additional mining locations.
- `OUTPUT_DIR` : Root directory for all pipeline outputs (datasets, models, detections).

Data Collection Settings

- `BRAZIL_BOUNDS` : Geographic bounding box to filter coordinates within Brazil.

- `N_FOREST_SAMPLES`: Number of negative (forest) samples to generate. Higher values improve model generalization but increase processing time.
- `DATE_RANGE`: Temporal window for satellite imagery queries.
- `MAX_CLOUD_COVER`: Maximum acceptable cloud coverage percentage.
- `IMAGE_SIZE_KM`: Spatial extent of each image patch (2.5 km from center = 5 km × 5 km area).

Dataset Building Settings

- `MATCH_TO_LANDSAT_STYLE`: Whether to apply color/contrast matching so Sentinel-2 images resemble Landsat imagery.
- `AUGMENTATION_STRENGTH`: Controls intensity of data augmentation ("light", "medium", "strong").
- `N_AUGMENTED_PER_IMAGE`: Number of augmented copies per original image.

Model Training Settings

- `BACKBONE`: CNN architecture ("resnet18", "resnet34", "efficientnet_b0").
- `BATCH_SIZE`, `LEARNING_RATE`, `EPOCHS`: Standard training hyperparameters.
- `EARLY_STOPPING_PATIENCE`: Number of epochs without improvement before stopping.
- `VALIDATION_SPLIT`: Fraction of data reserved for validation.
- `MODEL_IMAGE_SIZE`: Input resolution for the CNN (224 pixels for ResNet).

Inference Settings

- `MINING_THRESHOLD`: Probability threshold above which a prediction is classified as "mining".
- `OVERVIEW_RADIUS_KM`: Radius for the overview scan area in Step 5.

These parameters are referenced by Steps 1–6 and utility functions throughout the notebook.

In [ ]:
```
"""
Configuration parameters for the illegal mining detection pipeline.
"""

# ============================================================================
# DATA PATHS
# ============================================================================

# Local mine images (JPG files from Landsat) step 1
LOCAL_MINE_DIR = "GreenAI/src/data/landsat_converted/2019/barragem_jpg"

# CSV with mine coordinates (lon, lat columns) step 1
MINE_COORDINATES_CSV = "GreenAI/src/data/zones_centroids.csv"
```

```
# Output directory for all pipeline outputs
OUTPUT_DIR = "GreenAI/src/output"

# ============================================================================
# DATA COLLECTION SETTINGS
# ============================================================================

# Brazil bounding box (lon_min, lon_max, lat_min, lat_max)
BRAZIL_BOUNDS = (-75, -35, -35, 5)
# Number of forest (negative) samples to generate
N_FOREST_SAMPLES = 800  # amount of random samples to draw
# Satellite imagery settings (Landsat)
DATE_RANGE = "2023-01-01/2025-10-31"
MAX_CLOUD_COVER = 20
# Image size - distance from center point in km when retrieving data
IMAGE_SIZE_KM = 2.5

# ============================================================================
# DATASET BUILDING SETTINGS
# ============================================================================

# Match fetched images to local Landsat style
MATCH_TO_LANDSAT_STYLE = True #was created to evaluate if style transfer hel
# Augmentation settings
AUGMENTATION_STRENGTH = "medium"  # how many augmentations per image
N_AUGMENTED_PER_IMAGE = 5

# ============================================================================
# MODEL TRAINING SETTINGS
# ============================================================================

# Model architecture
BACKBONE = "resnet34"  # "resnet18", "resnet34", "efficientnet_b0"
HAS_SEGFORMER = True
HAS_TORCH = True
# Training hyperparameters
BATCH_SIZE = 16  # how many at once
LEARNING_RATE = 1e-4
EPOCHS = 300
EARLY_STOPPING_PATIENCE = 10
VALIDATION_SPLIT = 0.15

# Image size for model input  224 resnet
MODEL_IMAGE_SIZE = 224

# ============================================================================
# INFERENCE SETTINGS
# ============================================================================

# Confidence threshold for mining detection
MINING_THRESHOLD = 0.5

# ============================================================================
# VALIDATION SETTINGS
# ============================================================================
```

```
# Path to CSV with known mining coordinates for validation
# Expected columns: lat, lon, label (where label is "mining" or "forest")
VALIDATION_CSV = "GreenAI/src/data/known_mining_sites.csv"

# Default radius for overview (10km from center = 20km x 20km area)
OVERVIEW_RADIUS_KM = 10.0
```

Satellite Image Fetcher

This cell defines the `SatelliteFetcher` class, which was coded to take care of all interactions with Microsoft Planetary Computer to retrieve the Sentinel-2 satellite imagery. This is used in multiple steps. It is coded in satellite_fetcher.py in the implementation.

Documentation

`PROTECTED_AREAS` : A list of Brazilian protected areas (national parks, reserves) which we defined as zones from which to sample when building the negative class. Each area has a center coordinate and radius defining the sampling region.

`center_to_bbox()` : function that converts a center point (lat/lon) and radius (km) into a bounding box for we can actually use for the satelllite API. Was specifically coded to take care of adapting fro the latitude-dependent width of longitude degrees.

`SatelliteFetcher` Class:

- **`__init__()`** : Initializes the STAC client connection to Planetary Computer and configures query parameters (date range, cloud cover threshold, band selection for RGB).
- **`fetch_image()`** : Retrieves a single image patch centered on given coordinates. Contains code for Landsat simulation (cf presentation) (downsampling to match lower resolution), color normalization to target statistics, and cloud filtering.
- **`fetch_overview()`** : Retrieves a large, high-resolution image for regional scanning in Step 5. Does not apply Landsat simulation since SegFormer benefits from full resolution.
- **`_enhance_contrast()`** : Applies percentile-based contrast stretching (2nd to 98th percentile) to improve visual quality and normalize brightness across images.
- **`_normalize_to_target()`** : Matches image color distribution to reference statistics, enabling style transfer from Landsat to Sentinel-2 imagery.
- **`generate_forest_samples()`** : Batch generates negative samples by randomly sampling coordinates from protected areas and fetching imagery for each.

Usage in the code

- **Step 1**: Fetches additional mining samples from CSV coordinates via `fetch_image()` .

- **Step 2**: Generates forest samples via `generate_forest_samples()`.
- **Step 5**: Fetches overview imagery via `fetch_overview()` and individual grid cell patches via `fetch_image()`.
- **Step 6**: Fetches validation images for ground-truth coordinates via `fetch_image()`.

In [ ]:
```python
#satellite fetcher class
"""
Satellite image fetching from Microsoft Planetary Computer.
Uses Sentinel-2 but simulates Landsat-upscaled aesthetics to match local tra

Optimized for memory efficiency with large images.
"""

PROTECTED_AREAS = [
    {"name": "Tumucumaque", "lat": 1.5, "lon": -52.5, "radius_km": 100},
    {"name": "Jaú", "lat": -2.0, "lon": -63.0, "radius_km": 80},
    {"name": "Mamirauá", "lat": -2.5, "lon": -65.0, "radius_km": 50},
    {"name": "Terra do Meio", "lat": -5.5, "lon": -53.0, "radius_km": 100},
    {"name": "Xingu", "lat": -10.5, "lon": -52.5, "radius_km": 100},
]


def center_to_bbox(
    center_lat: float,
    center_lon: float,
    radius_km: float
) -> Tuple[float, float, float, float]:
    """
    Convert center coordinate + radius to bounding box.

    Args:
        center_lat: Latitude of center point
        center_lon: Longitude of center point
        radius_km: Distance from center to edge in km

    Returns:
        (lon_min, lat_min, lon_max, lat_max)
    """
    # Latitude: 1 degree ≈ 111 km
    lat_offset = radius_km / 111.0

    # Longitude: depends on latitude (narrower near poles)
    lon_offset = radius_km / (111.0 * np.cos(np.radians(center_lat)))

    return (
        center_lon - lon_offset,  # lon_min
        center_lat - lat_offset,  # lat_min
        center_lon + lon_offset,  # lon_max
        center_lat + lat_offset   # lat_max
    )


class SatelliteFetcher:
    def __init__(
```

```
        self,
        date_range: str = "2023-01-01/2024-12-31",
        max_cloud_cover: int = 20
    ):
        self.collection = "sentinel-2-l2a"
        self.date_range = date_range
        self.max_cloud_cover = max_cloud_cover

        self.client = Client.open(
            "https://planetarycomputer.microsoft.com/api/stac/v1",
            modifier=pc.sign_inplace
        )
        self.bands = ["B04", "B03", "B02"]

    def fetch_image(
        self,
        lat: float,
        lon: float,
        distance_km: float = 2.5,
        target_stats: Optional[Dict] = None,
        output_size: int = 512,
        simulate_landsat: bool = True
    ) -> Tuple[Optional[np.ndarray], Dict]:
        """
        Fetch a satellite image centered on a point.

        Args:
            lat: Latitude of center point
            lon: Longitude of center point
            distance_km: Distance from center to edge in km
            target_stats: Color statistics to match (optional)
            output_size: Output image size in pixels
            simulate_landsat: If True, applies blur to match Landsat-upscale

        Returns:
            (RGB image as numpy array, metadata dict)
        """
        bbox = center_to_bbox(lat, lon, distance_km)

        try:
            search = self.client.search(
                collections=[self.collection],
                bbox=bbox,
                datetime=self.date_range,
                query={"eo:cloud_cover": {"lt": self.max_cloud_cover}}
            )

            items = list(search.items())
            if not items:
                return None, {"error": "No imagery found", "lat": lat, "lon"

            items.sort(key=lambda x: x.properties.get("eo:cloud_cover", 100)
            item = items[0]

            rgb_bands = []
            for band_name in self.bands:
```

```
                href = item.assets[band_name].href
                with rasterio.open(href) as src:
                    src_bbox = transform_bounds('EPSG:4326', src.crs, *bbox)
                    window = src.window(*src_bbox)
                    data = src.read(1, window=window)
                    if data.size == 0:
                        return None, {"error": "Empty", "lat": lat, "lon": l
                    rgb_bands.append(data.astype(np.float32))

            # Align shapes
            min_h = min(b.shape[0] for b in rgb_bands)
            min_w = min(b.shape[1] for b in rgb_bands)
            rgb_bands = [b[:min_h, :min_w] for b in rgb_bands]

            rgb = np.stack(rgb_bands, axis=-1)
            del rgb_bands  # Free memory

            # Normalization (in-place where possible)
            np.clip(rgb, 0, 10000, out=rgb)
            rgb /= 10000.0
            rgb_enhanced = self._enhance_contrast(rgb)
            del rgb

            if target_stats:
                rgb_final = self._normalize_to_target(rgb_enhanced, target_s
                del rgb_enhanced
            else:
                rgb_final = rgb_enhanced

            rgb_uint8 = (np.clip(rgb_final, 0, 1) * 255).astype(np.uint8)
            del rgb_final

            img = Image.fromarray(rgb_uint8)
            del rgb_uint8

            if simulate_landsat:
                low_res_size = 93
                img_small = img.resize((low_res_size, low_res_size), Image.B
                img = img_small.resize((output_size, output_size), Image.BIC
                platform_note = "Sentinel-2 (Downsampled to match Landsat)"
            else:
                img = img.resize((output_size, output_size), Image.LANCZOS)
                platform_note = "Sentinel-2 (Full Resolution)"

            metadata = {
                "lat": lat, "lon": lon,
                "platform": platform_note,
                "cloud_cover": item.properties.get("eo:cloud_cover", None),
                "datetime": item.properties.get("datetime", None)
            }

            return np.array(img), metadata

        except Exception as e:
            return None, {"error": str(e), "lat": lat, "lon": lon}
```

```
    def fetch_overview(
        self,
        center_lat: float,
        center_lon: float,
        radius_km: float = 10.0,
        max_dimension: int = 2048,
        target_stats: Optional[Dict] = None
    ) -> Tuple[Optional[np.ndarray], Dict]:
        """
        Fetch a HIGH-RESOLUTION overview image centered on a point.

        This method fetches Sentinel-2 imagery at full resolution (no blur s
        for use with SegFormer segmentation.

        Args:
            center_lat: Latitude of center point
            center_lon: Longitude of center point
            radius_km: Distance from center to edge in km (default 10km = 20
            max_dimension: Maximum width or height of output image
            target_stats: Color statistics to match (optional)

        Returns:
            (RGB image as numpy array, metadata dict with bounds info)
        """
        bbox = center_to_bbox(center_lat, center_lon, radius_km)
        lon_min, lat_min, lon_max, lat_max = bbox

        lat_span_km = (lat_max - lat_min) * 111.0
        lon_span_km = (lon_max - lon_min) * 111.0 * np.cos(np.radians(center

        print(f"    Fetching overview: {lon_span_km:.1f} km x {lat_span_km:.
        print(f"    Center: ({center_lat:.4f}, {center_lon:.4f}), Radius: {r

        try:
            search = self.client.search(
                collections=[self.collection],
                bbox=list(bbox),
                datetime=self.date_range,
                query={"eo:cloud_cover": {"lt": self.max_cloud_cover}}
            )

            items = list(search.items())
            if not items:
                return None, {"error": "No imagery found", "bbox": bbox}

            items.sort(key=lambda x: x.properties.get("eo:cloud_cover", 100)
            item = items[0]

            print(f"    Found {len(items)} images, using best with {item.pro

            # Load all bands and determine actual shapes from data
            rgb_bands = []
            native_shape = None

            for band_name in self.bands:
                href = item.assets[band_name].href
```

```
                with rasterio.open(href) as src:
                    src_bbox = transform_bounds('EPSG:4326', src.crs, *bbox)
                    window = src.window(*src_bbox)
                    data = src.read(1, window=window)

                    if data.size == 0:
                        return None, {"error": "Empty window", "bbox": bbox}

                    if native_shape is None:
                        native_shape = data.shape
                        print(f"    Native resolution: {data.shape[1]}x{data

                    rgb_bands.append(data.astype(np.float32))
                    del data

                gc.collect()

            # Use minimum dimensions across bands (they can differ slightly)
            min_h = min(b.shape[0] for b in rgb_bands)
            min_w = min(b.shape[1] for b in rgb_bands)

            # Stack into single array, trimming to common dimensions
            rgb = np.zeros((min_h, min_w, 3), dtype=np.float32)
            for i, band in enumerate(rgb_bands):
                rgb[:, :, i] = band[:min_h, :min_w]

            del rgb_bands
            gc.collect()

            # In-place normalization to save memory
            np.clip(rgb, 0, 10000, out=rgb)
            rgb /= 10000.0

            # Enhance contrast (returns new array, but we delete old immedia
            rgb_enhanced = self._enhance_contrast(rgb)
            del rgb
            gc.collect()

            if target_stats:
                rgb_final = self._normalize_to_target(rgb_enhanced, target_s
                del rgb_enhanced
            else:
                rgb_final = rgb_enhanced

            # Convert to uint8 (final output format)
            rgb_uint8 = (np.clip(rgb_final, 0, 1) * 255).astype(np.uint8)
            del rgb_final
            gc.collect()

            # Resize if needed
            h, w = rgb_uint8.shape[:2]
            if max(h, w) > max_dimension:
                scale = max_dimension / max(h, w)
                new_w = int(w * scale)
                new_h = int(h * scale)
                img = Image.fromarray(rgb_uint8)
```

```
                del rgb_uint8
                img = img.resize((new_w, new_h), Image.LANCZOS)
                rgb_uint8 = np.array(img)
                del img
                print(f"    Resized to: {new_w}x{new_h} pixels")

            final_h, final_w = rgb_uint8.shape[:2]
            meters_per_pixel_x = (lon_span_km * 1000) / final_w
            meters_per_pixel_y = (lat_span_km * 1000) / final_h

            metadata = {
                "bbox": bbox,
                "center_lat": center_lat,
                "center_lon": center_lon,
                "radius_km": radius_km,
                "width_km": lon_span_km,
                "height_km": lat_span_km,
                "image_width": final_w,
                "image_height": final_h,
                "meters_per_pixel": (meters_per_pixel_x + meters_per_pixel_y
                "platform": "Sentinel-2 (Full Resolution)",
                "cloud_cover": item.properties.get("eo:cloud_cover", None),
                "datetime": item.properties.get("datetime", None),
                "native_resolution": native_shape
            }

            return rgb_uint8, metadata

        except Exception as e:
            import traceback
            traceback.print_exc()
            return None, {"error": str(e), "bbox": bbox}

    def _enhance_contrast(self, rgb: np.ndarray) -> np.ndarray:
        """Enhance contrast using percentile stretching."""
        result = np.zeros_like(rgb, dtype=np.float32)
        for i in range(3):
            band = rgb[:, :, i]
            valid = band[band > 0.001]
            if len(valid) > 0:
                p2, p98 = np.percentile(valid, [2, 98])
                if p98 > p2:
                    result[:, :, i] = (band - p2) / (p98 - p2)
                else:
                    result[:, :, i] = band
            else:
                result[:, :, i] = band
        return np.clip(result, 0, 1)

    def _normalize_to_target(self, rgb: np.ndarray, target_stats: Dict) -> r
        """Normalize colors to match target statistics."""
        result = np.zeros_like(rgb, dtype=np.float32)
        for i, channel in enumerate(["r", "g", "b"]):
            band = rgb[:, :, i]
            valid = band[band > 0.001]
            if len(valid) > 0:
```

```
                src_p2, src_p98 = np.percentile(valid, [2, 98])
                tgt_p2 = target_stats.get(f"{channel}_p2", 0.0)
                tgt_p98 = target_stats.get(f"{channel}_p98", 1.0)
                if src_p98 > src_p2:
                    normalized = (band - src_p2) / (src_p98 - src_p2)
                    result[:, :, i] = normalized * (tgt_p98 - tgt_p2) + tgt_
                else:
                    result[:, :, i] = band
            else:
                result[:, :, i] = band
        return np.clip(result, 0, 1)

    def generate_forest_samples(
        self,
        n_samples: int,
        output_dir: str,
        target_stats: Optional[Dict] = None,
        brazil_bounds: Tuple[float, float, float, float] = (-75, -35, -35, 5
        distance_km: float = 2.5
    ) -> List[Dict]:
        """Generate random forest samples from protected areas."""
        output_path = Path(output_dir)
        output_path.mkdir(parents=True, exist_ok=True)

        samples = []
        attempts = 0
        max_attempts = n_samples * 10

        print(f"    Targeting {n_samples} samples (simulating upscaled quali

        while len(samples) < n_samples and attempts < max_attempts:
            attempts += 1
            area = random.choice(PROTECTED_AREAS)
            offset_km = random.uniform(0, area["radius_km"])
            angle = random.uniform(0, 2 * np.pi)
            lat = area["lat"] + (offset_km / 111) * np.sin(angle)
            lon = area["lon"] + (offset_km / 111) * np.cos(angle)

            rgb, metadata = self.fetch_image(lat, lon, distance_km=distance_

            if rgb is not None:
                mean_val = rgb.mean()
                if 20 < mean_val < 230:
                    filename = f"forest_{len(samples):04d}.jpg"
                    filepath = output_path / filename
                    Image.fromarray(rgb).save(filepath, quality=95)
                    metadata["filename"] = filename
                    samples.append(metadata)
                    if len(samples) % 10 == 0:
                        print(f"    [+] Generated {len(samples)}/{n_samples}

                del rgb
                gc.collect()

        with open(output_path / "metadata.json", "w") as f:
```

```
            json.dump(samples, f, indent=2)
        return samples
```

Statistics Extraction

`extract_statistics()` is a utility function that computes color distribution statistics from a directory of images. These statistics are used to normalize our iages between different satellite imagery sources. **we had a problem when merging the different datasets together so we had to come up with a few solutions (manual editing of the parameters etc, normalization etc).**

Purpose

The training data consists of Landsat imagery with specific color characteristics. When fetching new images from Sentinel-2 (which has different radiometric properties), we needed to transform them to match the Landsat "style" (dataset we found online) so the classifier generalizes properly. This function extracts the target distribution that new images should be matched to.

Implementation Details

Due to multiple memory management problems we encountered, we eventually found the solution of using reservoir sampling to efficiently process large image collections without loading everything into memory. It can sample up to 100,000 pixels and computes per-channel (R, G, B) statistics including mean, standard deviation, and the 2nd/98th percentiles.

Output Statistics

- `r_mean`, `g_mean`, `b_mean`: Mean values for each channel
- `r_std`, `g_std`, `b_std`: Standard deviations
- `r_p2`, `g_p2`, `b_p2`: 2nd percentile values (dark reference)
- `r_p98`, `g_p98`, `b_p98`: 98th percentile values (bright reference)

Usage in Pipeline

- **Step 1**: Extracts statistics from local Landsat mine images immediately after copying them.
- **Step 3**: Uses these statistics via `match_image_to_stats()` to transform forest images during dataset building.
- **Steps 5 & 6**: Passes statistics to `SatelliteFetcher` for real-time style matching during inference.

In [ ]:
```python
def extract_statistics(image_dir: str, sample_pixels: int = 100000) -> Dict:
    """
    Extract RGB statistics from a directory of images.
    Uses reservoir sampling for memory efficiency.

    Args:
        image_dir: Directory containing JPG/PNG images
        sample_pixels: Max pixels to sample (default 100k uses ~1.2MB)

    Returns:
        Dictionary with per-channel statistics
    """
    image_path = Path(image_dir)
    image_files = list(image_path.glob("*.jpg")) + list(image_path.glob("*.p

    if not image_files:
        raise ValueError(f"No images found in {image_dir}")

    # Pre-allocate fixed-size reservoir (memory-efficient)
    reservoir = np.zeros((sample_pixels, 3), dtype=np.float32)
    total_seen = 0

    for img_file in image_files:
        img = np.array(Image.open(img_file).convert("RGB"), dtype=np.float32
        pixels = img.reshape(-1, 3)  # Flatten to (N, 3)

        for px in pixels:
            if total_seen < sample_pixels:
                reservoir[total_seen] = px
            else:
                # Reservoir sampling: replace with probability sample_pixels
                j = random.randint(0, total_seen)
                if j < sample_pixels:
                    reservoir[j] = px
            total_seen += 1

        del img, pixels  # Free memory

    # Use only filled portion
    n = min(total_seen, sample_pixels)
    samples = reservoir[:n]

    stats = {
        "r_mean": float(np.mean(samples[:, 0])),
        "r_std": float(np.std(samples[:, 0])),
        "r_p2": float(np.percentile(samples[:, 0], 2)),
        "r_p98": float(np.percentile(samples[:, 0], 98)),

        "g_mean": float(np.mean(samples[:, 1])),
        "g_std": float(np.std(samples[:, 1])),
        "g_p2": float(np.percentile(samples[:, 1], 2)),
        "g_p98": float(np.percentile(samples[:, 1], 98)),

        "b_mean": float(np.mean(samples[:, 2])),
        "b_std": float(np.std(samples[:, 2])),
```

```
        "b_p2": float(np.percentile(samples[:, 2], 2)),
        "b_p98": float(np.percentile(samples[:, 2], 98)),

        "n_images": len(image_files),
        "n_pixels_sampled": n,
        "source_dir": str(image_dir)
    }

    return stats
```

Now Comes the fun part. As mentioned in the presentation, we did not have an actually well built dataset to start with, reason being at the time of the project there were very little sources and the ones we actually did find were often times temporarily down. However, we eventually found a dataset of images containing legal mining dams. Taking what we had, we started our projects with this but this led to a whole bunch of other problems especially dataset uniformity.

Step 1: Collecting Mine Images (Positive Samples)

This step assembles the positive training samples representing known mining sites. It combines two data sources to maximize coverage.

Data Sources

Local Landsat Images: Pre-existing imagery from the `LOCAL_MINE_DIR` directory (Those of the dams we found online). The function only keeps the `*_512.jpg` files (512×512 pixel crops) and falls back to all JPG/PNG files if none are found. These images define the visual style that all other imagery will be matched to.

CSV Coordinates: Additional mining locations from `MINE_COORDINATES_CSV`. For each coordinate within Brazil bounds, the function fetches Sentinel-2 imagery via `SatelliteFetcher.fetch_image()`, applying Landsat style matching to ensure consistency with our dam images.

Process

1. Scans local directory for existing mine images
2. Copies local images to output directory with standardized naming (`mine_local_XXXX.jpg`)
3. Extracts color statistics from copied images (saved to `landsat_stats.json`)
4. Loads coordinates from CSV, filtering to Brazil bounding box
5. Fetches satellite imagery for each coordinate, applying style matching
6. Saves fetched images as `mine_fetched_XXXX.jpg`

Outputs

- `OUTPUT_DIR/raw_data/mines/` : Directory containing all positive samples
- `OUTPUT_DIR/landsat_stats.json` : Color statistics for style matching
- Returns dictionary with counts and statistics for use in subsequent steps

In [ ]:
```
# ============================================================================
# STEP 1: Collect Mine Images (Positive Samples)
# ============================================================================

def step1_collect_mines() -> Dict:
    print("\n" + "=" * 60)
    print("STEP 1: Collecting Mine Images (Positive Samples)")
    print("=" * 60)

    output_path = Path(OUTPUT_DIR) / "raw_data" / "mines"
    if output_path.exists():
        shutil.rmtree(output_path)
    output_path.mkdir(parents=True, exist_ok=True)

    local_path = Path(LOCAL_MINE_DIR).resolve()
    csv_path = Path(MINE_COORDINATES_CSV).resolve()

    print(f"  Local mine directory: {local_path}")

    local_images = []
    local_image_size = None

    if local_path.exists():
        print("  Scanning for *_512.jpg files...")
        local_images = list(local_path.glob("*_512.jpg"))

        if not local_images:
            print("  ⚠ No *_512.jpg files found. Falling back to all *.jpg f
            local_images = list(local_path.glob("*.jpg")) + list(local_path.
        else:
            print(f"  Found {len(local_images)} specific 512px images.")

        if local_images:

            first_img = Image.open(local_images[0])
            local_image_size = first_img.size
            print(f"  Local image size: {local_image_size[0]}x{local_image_s

    if local_images:
        print(f"  Copying {len(local_images)} local images to output...")
        for i, img_file in enumerate(local_images):
            dst_name = f"mine_local_{i:04d}.jpg"
            shutil.copy(img_file, output_path / dst_name)
            if (i + 1) % 100 == 0:
                print(f"    Copied {i + 1}/{len(local_images)}")
    else:
        print("  ⚠ No local images found!")

    landsat_stats = None
    if local_images:
        print("  Extracting color statistics from selected images...")
```

```
        landsat_stats = extract_statistics(str(output_path))

        stats_file = Path(OUTPUT_DIR) / "landsat_stats.json"
        with open(stats_file, "w") as f:
            json.dump(landsat_stats, f, indent=2)
        print(f"  Saved statistics to {stats_file}")

    fetched_count = 0
    coordinates = []

    if csv_path.exists():
        print(f"\n  Loading coordinates from CSV: {csv_path}")
        with open(csv_path, "r") as f:
            reader = csv.DictReader(f)
            lat_col = next((col for col in reader.fieldnames if col.lower()
            lon_col = next((col for col in reader.fieldnames if col.lower()

            if lat_col and lon_col:
                f.seek(0)
                reader = csv.DictReader(f)
                for row in reader:
                    try:
                        lat, lon = float(row[lat_col]), float(row[lon_col])
                        b = BRAZIL_BOUNDS
                        if not (b[0] < lon < b[1] and b[2] < lat < b[3]):
                            continue
                        coordinates.append({"lat": lat, "lon": lon})
                    except ValueError:
                        continue

    if coordinates:
        print(f"  Fetching {len(coordinates)} additional samples from CSV co

        try:


            fetcher = SatelliteFetcher(date_range=DATE_RANGE, max_cloud_cove

            for i, coord in enumerate(coordinates):
                rgb, metadata = fetcher.fetch_image(
                    lat=coord["lat"], lon=coord["lon"],
                    distance_km=IMAGE_SIZE_KM, target_stats=landsat_stats
                )

                if rgb is not None:
                    filename = f"mine_fetched_{fetched_count:04d}.jpg"
                    Image.fromarray(rgb).save(output_path / filename, qualit
                    fetched_count += 1

                if (i + 1) % 10 == 0:
                    print(f"    Processed {i + 1}/{len(coordinates)} ({fetch

                time.sleep(0.5)

        except ImportError as e:
            print(f"  ⚠ Could not fetch satellite images: {e}")
```

```
    result = {
        "output_dir": str(output_path),
        "local_count": len(local_images),
        "fetched_count": fetched_count,
        "total_count": len(local_images) + fetched_count,
        "landsat_stats": landsat_stats,
        "image_size": local_image_size
    }

    print(f"\n  Total mine images collected: {result['total_count']}")
    return result
```

Step 2: Collecting Forest Images (Negative Samples)

This step generates negative training samples representing only forest areas that should not be classified as mining.

Sampling Strategy

Rather than randomly sampling anywhere in Brazil (which might accidentally capture roads, cities, or small mines), this step specifically targets protected areas where deforestation and mining are prohibited. The `PROTECTED_AREAS` list defines five large conservation units in the Amazon which we found online after some reading.

For each sample, the function randomly selects a protected area, then picks a random point within that area's radius. This ensures geographic diversity while maintaining high confidence that the samples represent undisturbed forest.

Quality Filtering

After fetching each image, a basic quality check filters out problematic samples. Images with very low mean brightness (< 20) likely contain clouds or shadows, while very high brightness (> 230) may indicate over-exposure or data errors. Only images passing this check were saved.

Style Matching

All forest images are fetched with `target_stats` set to the Landsat statistics from Step 1. This allows us to make sure that the negative samples have the same color distribution as the positive samples. Yhis method was to prevent the classifier from learning to distinguish mining based on image style rather than actual content.

Outputs

- `OUTPUT_DIR/raw_data/forest/` : Directory containing all negative samples
- `metadata.json` : Coordinate and acquisition metadata for each sample
- Returns dictionary with sample count for use in Step 3

In [ ]:
```
# ============================================================================
# STEP 2: Collect Forest Images (Negative Samples)
# ============================================================================

def step2_collect_forest(landsat_stats: Dict) -> Dict:
    print("\n" + "=" * 60)
    print("STEP 2: Collecting Forest Images (Negative Samples)")
    print("=" * 60)

    output_path = Path(OUTPUT_DIR) / "raw_data" / "forest"
    output_path.mkdir(parents=True, exist_ok=True)

    try:

        fetcher = SatelliteFetcher(date_range=DATE_RANGE, max_cloud_cover=MA

        print(f"  Generating {N_FOREST_SAMPLES} forest samples...")

        samples = fetcher.generate_forest_samples(
            n_samples=N_FOREST_SAMPLES,
            output_dir=str(output_path),
            target_stats=landsat_stats,
            brazil_bounds=BRAZIL_BOUNDS,
            distance_km=IMAGE_SIZE_KM
        )

        result = {"output_dir": str(output_path), "count": len(samples)}
        print(f"\n  Generated {len(samples)} forest images")

    except ImportError as e:
        print(f"  Warning: Could not fetch forest images: {e}")
        result = {"output_dir": str(output_path), "count": 0}

    return result
```

Dataset Builder Utility Functions

Here we defined helper functions and classes for transforming the retrieved imagesg.

`match_image_to_stats()`

Transforms a single image to match target color statistics. Uses percentile-based histogram matching: the source image's 2nd–98th percentile range is linearly mapped to the target's corresponding range for each RGB channel. This is applied to forest images during dataset building to ensure they match the Landsat visual style.

`SatelliteAugmentation` Class

Implements domain-appropriate data augmentation for satellite imagery. Unlike general-purpose augmentation (which might use arbitrary rotations or elastic deformations), this class applies only transformations that are physically realistic for overhead satellite views:

- **90° rotations**: Satellites can image from different orbit passes; arbitrary angles would create unrealistic orientations.
- **Horizontal/vertical flips**: Equally valid viewing directions.
- **Brightness adjustment**: Simulates varying sun angles and atmospheric conditions.
- **Contrast adjustment**: Simulates atmospheric haze density variations.
- **Atmospheric haze simulation**: Adds luminance-dependent brightening that mimics real haze effects.
- **Sensor noise**: Adds Gaussian noise to simulate sensor imperfections.

The `strength` parameter ("light", "medium", "strong") controls the magnitude of all transformations. This was implemented after multiple iterations. We could've removed the parameters but it did not change much.

`build_dataset()`

taulches the complete dataset construction process:

1. Creates train/validation directory structure
2. Shuffles and splits source images (default 85% train, 15% validation)
3. Copies original images to appropriate directories
4. Applies distribution matching to negative samples
5. Generates augmented versions for training set only (validation uses clean originals)
6. Saves metadata JSON with counts and statistics

`merge_image_directories()`

Utility function to combine images from our source directories into one.

In [ ]:
```
"""
helper functins for the extraction of statistics, matching distributions, ar
Dataset building: statistics extraction, distribution matching, and augmenta
"""


def match_image_to_stats(image: np.ndarray, target_stats: Dict) -> np.ndarra
    """
    Transform an image to match target statistics (Landsat style).

    Args:
        image: RGB image as numpy array (uint8, 0-255)
        target_stats: Statistics from extract_statistics()
```

```
    Returns:
        Matched image as numpy array (uint8, 0-255)
    """
    img = image.astype(np.float32) / 255.0
    result = np.zeros_like(img)

    for i, channel in enumerate(["r", "g", "b"]):
        src = img[:, :, i]

        # Get source percentiles
        src_p2, src_p98 = np.percentile(src, [2, 98])

        # Get target percentiles
        tgt_p2 = target_stats[f"{channel}_p2"]
        tgt_p98 = target_stats[f"{channel}_p98"]

        # Map source to target distribution
        if src_p98 > src_p2:
            normalized = (src - src_p2) / (src_p98 - src_p2)
            result[:, :, i] = normalized * (tgt_p98 - tgt_p2) + tgt_p2
        else:
            result[:, :, i] = target_stats[f"{channel}_mean"]

    return (np.clip(result, 0, 1) * 255).astype(np.uint8)


class SatelliteAugmentation:
    """
    Domain-appropriate augmentation for satellite imagery.

    Satellite images need:
    - Only 90° rotations (not arbitrary angles)
    - Atmospheric haze simulation
    - Sensor noise
    - No elastic deformation
    """

    def __init__(self, strength: str = "medium"):
        """
        Args:
            strength: "light", "medium", or "strong"
        """
        self.strength = strength

        # Parameters per strength level
        self.params = {
            "light": {
                "brightness_range": (0.95, 1.05),
                "contrast_range": (0.95, 1.05),
                "noise_std": 0.01,
                "haze_prob": 0.1,
                "haze_strength": 0.05
            },
            "medium": {
                "brightness_range": (0.85, 1.15),
```

```
                "contrast_range": (0.85, 1.15),
                "noise_std": 0.02,
                "haze_prob": 0.3,
                "haze_strength": 0.1
            },
            "strong": {
                "brightness_range": (0.7, 1.3),
                "contrast_range": (0.7, 1.3),
                "noise_std": 0.03,
                "haze_prob": 0.5,
                "haze_strength": 0.15
            }
        }[strength]

    def augment(self, image: np.ndarray) -> np.ndarray:
        """Apply random augmentation to an image."""
        img = image.astype(np.float32) / 255.0

        # 1. Random 90° rotation
        k = random.randint(0, 3)
        img = np.rot90(img, k)

        # 2. Random flip
        if random.random() < 0.5:
            img = np.fliplr(img)
        if random.random() < 0.5:
            img = np.flipud(img)

        # 3. Brightness adjustment (sun angle simulation)
        brightness = random.uniform(*self.params["brightness_range"])
        img = img * brightness

        # 4. Contrast adjustment
        contrast = random.uniform(*self.params["contrast_range"])
        mean = img.mean()
        img = (img - mean) * contrast + mean

        # 5. Atmospheric haze
        if random.random() < self.params["haze_prob"]:
            haze = self.params["haze_strength"] * random.random()
            # Haze is more visible in darker areas
            luminance = 0.299 * img[:, :, 0] + 0.587 * img[:, :, 1] + 0.114
            haze_mask = 1 - luminance
            for i in range(3):
                img[:, :, i] = img[:, :, i] + haze * haze_mask

        # 6. Sensor noise
        noise = np.random.normal(0, self.params["noise_std"], img.shape)
        img = img + noise

        return (np.clip(img, 0, 1) * 255).astype(np.uint8)

    def augment_batch(self, image: np.ndarray, n: int) -> List[np.ndarray]:
        """Generate n augmented versions of an image."""
        return [self.augment(image) for _ in range(n)]
```

```python
def build_dataset(
    positive_dir: str,
    negative_dir: str,
    output_dir: str,
    target_stats: Optional[Dict] = None,
    augmentation_strength: str = "medium",
    n_augmented_per_image: int = 5,
    match_distributions: bool = True,
    val_split: float = 0.2
) -> Dict:
    """
    Build a complete training dataset with a PHYSICAL Train/Validation split

    Structure:
        output_dir/
            train/
                positive/ (Originals + Augmented)
                negative/ (Originals + Augmented)
            val/
                positive/ (Originals only)
                negative/ (Originals only)
    """
    output_path = Path(output_dir)

    # 1. Prepare Directory Structure
    for split in ["train", "val"]:
        for label in ["positive", "negative"]:
            (output_path / split / label).mkdir(parents=True, exist_ok=True)

    # 2. Extract stats if needed
    if target_stats is None and match_distributions:
        print("  Extracting statistics from positive images...")
        target_stats = extract_statistics(positive_dir)

    augmenter = SatelliteAugmentation(strength=augmentation_strength)

    metadata = {
        "train_positive": 0, "train_negative": 0,
        "val_positive": 0, "val_negative": 0,
        "target_stats": target_stats
    }

    # 3. Processing Logic
    def process_class(source_dir, class_name, is_positive):
        files = list(Path(source_dir).glob("*.jpg")) + list(Path(source_dir)

        # SHUFFLE AND SPLIT
        random.shuffle(files)
        split_idx = int(len(files) * (1 - val_split))
        train_files = files[:split_idx]
        val_files = files[split_idx:]

        print(f"  Processing {class_name}: {len(train_files)} training, {ler
```

```
        # --- TRAIN SET (Augmented) ---
        for i, img_file in enumerate(train_files):
            img = np.array(Image.open(img_file).convert("RGB"))

            # Distribution matching (for Forest images)
            if not is_positive and match_distributions and target_stats:
                img = match_image_to_stats(img, target_stats)

            # Save Original
            base_name = f"{class_name}_{i:05d}"
            Image.fromarray(img).save(output_path / "train" / class_name / f
            metadata[f"train_{class_name}"] += 1

            # Save Augmented versions
            for j in range(n_augmented_per_image):
                aug_img = augmenter.augment(img)
                aug_name = f"{base_name}_aug{j:02d}.jpg"
                Image.fromarray(aug_img).save(output_path / "train" / class_
                metadata[f"train_{class_name}"] += 1
                del aug_img
            del img

        # --- VAL SET ---
        for i, img_file in enumerate(val_files):
            img = np.array(Image.open(img_file).convert("RGB"))

            # Still apply distribution matching to validation negatives so t
            if not is_positive and match_distributions and target_stats:
                img = match_image_to_stats(img, target_stats)

            # Save Original Only
            val_name = f"{class_name}_{i:05d}.jpg"
            Image.fromarray(img).save(output_path / "val" / class_name / val
            metadata[f"val_{class_name}"] += 1
            del img

    # 4. Execute
    print("  Processing Positive (Mine) images...")
    process_class(positive_dir, "positive", is_positive=True)

    print("  Processing Negative (Forest) images...")
    process_class(negative_dir, "negative", is_positive=False)

    # 5. Save Metadata
    json_metadata = {k: v for k, v in metadata.items() if k != "target_stats
    if target_stats:
        json_metadata["target_stats"] = target_stats

    with open(output_path / "dataset_metadata.json", "w") as f:
        json.dump(json_metadata, f, indent=2)

    return metadata
```

```python
def merge_image_directories(dirs: List[str], output_dir: str) -> int:
    """
    Merge multiple image directories into one.

    Args:
        dirs: List of source directories
        output_dir: Output directory

    Returns:
        Number of images copied
    """
    output_path = Path(output_dir)
    output_path.mkdir(parents=True, exist_ok=True)

    count = 0
    for src_dir in dirs:
        src_path = Path(src_dir)
        if not src_path.exists():
            print(f"  Warning: {src_dir} does not exist, skipping")
            continue

        for img_file in list(src_path.glob("*.jpg")) + list(src_path.glob("*
            dst_name = f"img_{count:04d}{img_file.suffix}"
            shutil.copy(img_file, output_path / dst_name)
            count += 1

    return count
```

Step 3: Build Training Dataset

Here we transformed the raw collected images into a properly structured dataset for the neural network training.

Dataset Structure

The previous execution saved the images to:

```
OUTPUT_DIR/dataset/
├── train/
│   ├── positive/    # Mining images + augmented versions
│   └── negative/    # Forest images + augmented versions
└── val/
    ├── positive/    # Mining images (originals only)
    └── negative/    # Forest images (originals only)
```

Design Decisions

Physical Train/Val Split: Images are split before augmentation, ensuring that augmented versions of validation images never appear in training. This prevents data leakage that would inflate validation metrics.

Augmentation on Training Only: The validation set contains only original images to provide an unbiased estimate of real-world performance. Training images receive multiple augmented copies to increase effective dataset size and improve generalization.

Distribution Matching: All fetched images are transformed to match the mining dam image statistics. This forces the classifier to learn semantic features (disturbed land patterns) instead of (image level) differences in image style.

Outputs

- `OUTPUT_DIR/dataset/` : Complete dataset directory
- `dataset_metadata.json` : Counts for train/val positive/negative splits
- Returns metadata dictionary we used for verification

In [ ]:
```
# ============================================================================
# STEP 3: Build Training Dataset
# ============================================================================

def step3_build_dataset(mines_dir: str, forest_dir: str, landsat_stats: Dict
    print("\n" + "=" * 60)
    print("STEP 3: Building Training Dataset (Split First -> Augment Train O
    print("=" * 60)

    output_path = Path(OUTPUT_DIR) / "dataset"


    metadata = build_dataset(
        positive_dir=mines_dir,
        negative_dir=forest_dir,
        output_dir=str(output_path),
        target_stats=landsat_stats,
        augmentation_strength=AUGMENTATION_STRENGTH,
        n_augmented_per_image=N_AUGMENTED_PER_IMAGE,
        match_distributions=MATCH_TO_LANDSAT_STYLE,
        val_split=VALIDATION_SPLIT
    )

    metadata["output_dir"] = str(output_path)
    print(f"\n  Dataset built at {output_path}")

    return metadata
```

CNN Classifier Components

Here we define the neural network architecture, dataset loader, training loop, and inference predictor for the mining classification model.

`MiningDataset` Class

A PyTorch Dataset that loads images from the train/val directory structure created in Step 3. Params:

- Accepts a `split` parameter ("train" or "val") to load from the appropriate subdirectory
- Applies standard ImageNet normalization (mean/std from ImageNet pretraining)
- Resizes all images to the model's expected input size (224×224 for ResNet)
- Returns (image_tensor, label) pairs where label is 1.0 for mining, 0.0 for forest

`MiningClassifier` Class

A CNN model that uses a pretrained backbone (Here we had tom implement multiple because sometimes or computer charshed bc not enough RAM ResNet-18, ResNet-34, or EfficientNet-B0) with a binary classification head. The idea being that pretrained ImageNet weights would give us already good low-level feature extractors. What we then did was train the final classifier layer, which we retrained from scratch (final layer is where often times the patterns are merged eg vertical line + horizontal line might mean 8 for the resnet but we want it to learn "hole" or something other) (we need to imagine the previous example in a much higher dimension but that was the best example I could come up with to explain ie a combination of many different low-leve features can represent "dog" or "dam" depending on the contecxt, which is taken care of in the final layer from what we understood)

We also set the forward pass to apply sigmoid activation to produce a probability in [0, 1]. **The idea was that mabye we could then choose the threshold probability manually for cases where the model had a hard time decidig**

`train_model()` Function

Contains the whole training loop with:

- Adam optimizer
- Binary cross-entropy loss for loss function (best function for bin class)
- Learning rate scheduling (ReduceLROnPlateau) that reduces LR when validation loss plateaus
- Early stopping to prevent overfitting
- Checkpoint saving for best model (by validation loss)
- Training history logging (loss and accuracy per epoch)

`Predictor` Class

wrapper that loads our trained model checkpoint (model) and allows us to predict:

- `predict(image_path)` : Classify a single image file

- `predict_array(image)` : Classify a numpy array (useful for in-memory images)
- `predict_batch(image_dir)` : Classify all images in a directory

All methods return a dictionary with probability, predicted class ("mining" or "forest"), and boolean flag.

In [ ]:
```
"""
CNN classifier for mining detection.
"""



class MiningDataset(Dataset):
    """
    PyTorch Dataset for mining classification.
    Loads from a specific split directory ('train' or 'val').
    """

    def __init__(
        self,
        dataset_dir: str,
        split: str = "train",  # Added split argument
        image_size: int = 224
    ):
        self.image_size = image_size
        self.split = split

        # Standard normalization for both train and val
        # (Augmentation is now done offline in build_dataset)
        self.transform = transforms.Compose([
            transforms.Resize((image_size, image_size)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.2
        ])

        root_path = Path(dataset_dir) / split
        if not root_path.exists():
            raise ValueError(f"Dataset split directory not found: {root_path

        self.samples = []

        # Load positive samples (Label: 1)
        pos_dir = root_path / "positive"
        if pos_dir.exists():
            for img_file in pos_dir.glob("*.jpg"):
                self.samples.append((str(img_file), 1.0))

        # Load negative samples (Label: 0)
        neg_dir = root_path / "negative"
        if neg_dir.exists():
            for img_file in neg_dir.glob("*.jpg"):
                self.samples.append((str(img_file), 0.0))

        if len(self.samples) == 0:
            raise ValueError(f"No images found in {root_path}")
```

```
    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        img_path, label = self.samples[idx]
        try:
            img = Image.open(img_path).convert("RGB")
            img_tensor = self.transform(img)
            return img_tensor, label
        except Exception as e:
            print(f"Error loading {img_path}: {e}")
            # Return a dummy tensor in case of corruption, or handle appropr
            return torch.zeros((3, self.image_size, self.image_size)), label



class MiningClassifier(nn.Module):
    """CNN classifier with pretrained backbone."""

    def __init__(self, backbone: str = "resnet18", pretrained: bool = True):
        super().__init__()

        if backbone == "resnet18":
            self.backbone = models.resnet18(pretrained=pretrained)
            num_features = self.backbone.fc.in_features
            self.backbone.fc = nn.Sequential(
                nn.Dropout(0.5),
                nn.Linear(num_features, 1)
            )
        elif backbone == "resnet34":
            self.backbone = models.resnet34(pretrained=pretrained)
            num_features = self.backbone.fc.in_features
            self.backbone.fc = nn.Sequential(
                nn.Dropout(0.5),
                nn.Linear(num_features, 1)
            )
        elif backbone == "efficientnet_b0":
            self.backbone = models.efficientnet_b0(pretrained=pretrained)
            num_features = self.backbone.classifier[1].in_features
            self.backbone.classifier = nn.Sequential(
                nn.Dropout(0.5),
                nn.Linear(num_features, 1)
            )
        else:
            raise ValueError(f"Unknown backbone: {backbone}")

    def forward(self, x):
        return torch.sigmoid(self.backbone(x))


def train_model(
    dataset_dir: str,
    output_dir: str,
    backbone: str = "resnet18",
    batch_size: int = 32,
```

```
    learning_rate: float = 1e-4,
    epochs: int = 30,
    patience: int = 7,
    image_size: int = 224
) -> Dict:
    """
    Train a mining classifier using pre-split folders.
    """
    if not HAS_TORCH:
        raise ImportError("PyTorch required for training")

    output_path = Path(output_dir)
    output_path.mkdir(parents=True, exist_ok=True)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"  Using device: {device}")

    # 1. Load Datasets (Physical Split)
    print("  Loading Training Set...")
    train_dataset = MiningDataset(dataset_dir, split="train", image_size=ima

    print("  Loading Validation Set...")
    val_dataset = MiningDataset(dataset_dir, split="val", image_size=image_s

    print(f"  Train samples: {len(train_dataset)}")
    print(f"  Val samples:   {len(val_dataset)}")

    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=Fals

    # 2. Model Setup
    model = MiningClassifier(backbone=backbone).to(device)
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=3,

    # 3. Training Loop
    history = {"train_loss": [], "val_loss": [], "val_acc": []}
    best_val_loss = float("inf")
    patience_counter = 0

    for epoch in range(epochs):
        # Train
        model.train()
        train_loss = 0.0

        for images, labels in train_loader:
            images = images.to(device)
            labels = labels.float().to(device).unsqueeze(1)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
```

```
            train_loss += loss.item() * images.size(0)

        train_loss /= len(train_dataset)

        # Validate
        model.eval()
        val_loss = 0.0
        correct = 0
        total = 0

        with torch.no_grad():
            for images, labels in val_loader:
                images = images.to(device)
                labels = labels.float().to(device).unsqueeze(1)

                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item() * images.size(0)

                predicted = (outputs > 0.5).float()
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        val_loss /= len(val_dataset)
        val_acc = correct / total

        # Scheduler Step
        scheduler.step(val_loss)

        # Logging
        history["train_loss"].append(train_loss)
        history["val_loss"].append(val_loss)
        history["val_acc"].append(val_acc)

        print(f"  Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} |

        # Early Stopping
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            patience_counter = 0
            torch.save(model.state_dict(), output_path / "best_model.pth")
            # print("    -> Saved best model")
        else:
            patience_counter += 1
            if patience_counter >= patience:
                print(f"  Early stopping triggered at epoch {epoch+1}")
                break

    # Save Final
    torch.save(model.state_dict(), output_path / "final_model.pth")

    with open(output_path / "training_history.json", "w") as f:
        json.dump(history, f, indent=2)

    return history
```

```python
class Predictor:
    """Run inference with a trained model."""

    def __init__(
        self,
        model_path: str,
        backbone: str = "resnet18",
        image_size: int = 224,
        threshold: float = 0.5
    ):
        if not HAS_TORCH:
            raise ImportError("PyTorch required for inference")

        self.device = torch.device("cuda" if torch.cuda.is_available() else
        self.threshold = threshold
        self.image_size = image_size

        # Load model
        self.model = MiningClassifier(backbone=backbone, pretrained=False)
        self.model.load_state_dict(torch.load(model_path, map_location=self.
        self.model.to(self.device)
        self.model.eval()

        # Transform
        self.transform = transforms.Compose([
            transforms.Resize((image_size, image_size)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.2
        ])

    def predict(self, image_path: str) -> Dict:
        """
        Predict on a single image.

        Returns:
            {"probability": float, "prediction": str, "is_mining": bool}
        """
        img = Image.open(image_path).convert("RGB")
        img_tensor = self.transform(img).unsqueeze(0).to(self.device)

        with torch.no_grad():
            prob = self.model(img_tensor).item()

        return {
            "probability": prob,
            "prediction": "mining" if prob > self.threshold else "forest",
            "is_mining": prob > self.threshold
        }

    def predict_array(self, image: np.ndarray) -> Dict:
        """Predict on a numpy array (RGB, 0-255)."""
        img = Image.fromarray(image)
        img_tensor = self.transform(img).unsqueeze(0).to(self.device)
```

```
        with torch.no_grad():
            prob = self.model(img_tensor).item()

        return {
            "probability": prob,
            "prediction": "mining" if prob > self.threshold else "forest",
            "is_mining": prob > self.threshold
        }

    def predict_batch(self, image_dir: str) -> List[Dict]:
        """Predict on all images in a directory."""
        results = []
        image_path = Path(image_dir)

        for img_file in list(image_path.glob("*.jpg")) + list(image_path.glo
            result = self.predict(str(img_file))
            result["filename"] = img_file.name
            results.append(result)

        return results
```

Step 4: Training the model

Herewe actually train the CNN classifier using the dataset prepared in Step 3.

Training Process

Creates a `MiningClassifier` object with the our backbone (resnet-34), it then loads train and validation datasets, and runs the training loop for up to `EPOCHS` iterations.

Training uses GPU acceleration (checked via `torch.cuda.is_available()`), but if not we can still run on cpu bit it mght kill your PC.

Note: The main problem we had in this whole project wass OOMs and memory management issues, which is why some code may seem ovely complicated. The DataLoader uses 4 worker processes and pinned memory for efficient data loading.

Checkpoints

During training, the function prints per-epoch metrics (training loss, validation loss, validation accuracy). The learning rate scheduler reduces LR by 50% if validation loss doesn't improve for 3 consecutive epochs. (early stopping, 3 was chosen later to explain)

Two model checkpoints are saved:

- `best_model.pth` : Weights from the epoch with lowest validation loss (used in inference)
- `final_model.pth` : Weights from the last training epoch

Early Stopping

If validation loss doesn't improve for `EARLY_STOPPING_PATIENCE` epochs, training terminates early to prevent overfitting. This is especially important given our relatively small dataset size. Too many epochs would lead to overfitting

Outputs

- `OUTPUT_DIR/model/best_model.pth` : Best model weights
- `OUTPUT_DIR/model/final_model.pth` : Final model weights
- `OUTPUT_DIR/model/training_history.json` : Loss and accuracy curves
- Returns history dictionary with metrics and model directory path

In [ ]:
```python
# ============================================================================
# STEP 4: Train Model
# ============================================================================

def step4_train(dataset_dir: str) -> Dict:
    print("\n" + "=" * 60)
    print("STEP 4: Training Classifier")
    print("=" * 60)

    output_path = Path(OUTPUT_DIR) / "model"

    history = train_model(
        dataset_dir=dataset_dir,
        output_dir=str(output_path),
        backbone=BACKBONE,
        batch_size=BATCH_SIZE,
        learning_rate=LEARNING_RATE,
        epochs=EPOCHS,
        patience=EARLY_STOPPING_PATIENCE,
        image_size=MODEL_IMAGE_SIZE
    )

    history["model_dir"] = str(output_path)
    print(f"\n  Model saved to {output_path}")

    return history
```

Segmentation Detector

As mentionned earlier, we mentionned the use of a SegFormer-based semantic segmentation model in a two stage pipeline (Step 5). In this cell we also putthe visualization utilities used in Step 5's two-stage detection pipeline.

`MiningSegmentationDetector` Class

Wraps a pretrained SegFormer model fine-tuned on the LoveDA dataset (a remote sensing land cover dataset https://github.com/Junjue-Wang/LoveDA, https://huggingface.co/wu-pr-gw/segformer-b2-finetuned-with-LoveDA). The model segments satellite imagery into seven classes:

Class ID	Label	Suspicious?
0	Background	No
1	Building	Yes
2	Road	Yes
3	Water	Yes
4	Barren	Yes
5	Forest	No
6	Agricultural	No

We considered Classes 1–4 should be flagged as "suspicious" because we assumed mining operations typically create barren land, often have associated buildings/roads, and frequently involve water (tailing ponds, river dredging).

Methods:

- `predict_mask()` : Runs the full segmentation pipeline, returning a class ID mask at the input image resolution.
- `get_suspicious_mask()` : Converts the class mask to a binary mask (suspicious vs. non-suspicious) and applies smoothing to reduce noise. (ie expands the regions a little bit)

`DetectionVisualizer` Class

CHAT GPT GENERATED: visualization utilities for debugging and output generation:

- `draw_segmentation_overlay()` : Renders the class mask as a color overlay on the original image.
- `draw_suspicious_overlay()` : Highlights suspicious regions in red.

Explanation

Using semantic segmentation as a first-stage filter dramatically reduces the search space. Rather than running the CNN classifier on every possible location, we only classify regions where the segmentation model has already identified potentially disturbed land.

In [ ]:
```
class MiningSegmentationDetector:
    """
    Robust SegFormer detector.
    Identifies 'suspicious' pixels including industrial/disturbed land.
```

```
    """

    # LoveDA Class Labels:
    # 0:Background, 1:Building, 2:Road, 3:Water, 4:Barren, 5:Forest, 6:Agric

    def __init__(
        self,
        model_name: str = "wu-pr-gw/segformer-b2-finetuned-with-LoveDA",
        device: Optional[str] = None
    ):
        if not HAS_SEGFORMER:
            raise ImportError("transformers required: pip install transforme

        self.device = device or ("cuda" if torch.cuda.is_available() else "c
        print(f"Loading SegFormer: {model_name}")

        try:
            self.processor = SegformerImageProcessor.from_pretrained(model_r
            self.model = SegformerForSemanticSegmentation.from_pretrained(mo
        except Exception as e:
            print(f"Warning: {e}. Fallback to generic model.")
            fallback = "nvidia/segformer-b0-finetuned-ade-512-512"
            self.processor = SegformerImageProcessor.from_pretrained(fallbac
            self.model = SegformerForSemanticSegmentation.from_pretrained(fa

        self.model.to(self.device)
        self.model.eval()

       # clsses are Building, Road, Water, Barren
        self.suspicious_classes = {1, 2, 3, 4}

    def predict_mask(self, image: np.ndarray) -> np.ndarray:
        """Returns the raw class ID mask (H, W)."""
        h, w = image.shape[:2]

        max_dim = 2048
        if max(h, w) > max_dim:
            scale = max_dim / max(h, w)
            new_w, new_h = int(w * scale), int(h * scale)
            img_input = cv2.resize(image, (new_w, new_h))
        else:
            img_input = image

        inputs = self.processor(images=img_input, return_tensors="pt")
        inputs = {k: v.to(self.device) for k, v in inputs.items()}

        with torch.no_grad():
            outputs = self.model(**inputs)
            logits = torch.nn.functional.interpolate(
                outputs.logits, size=(h, w), mode="bilinear", align_corners=
            )
            pred_mask = logits.argmax(dim=1).squeeze().cpu().numpy()

        if torch.cuda.is_available(): torch.cuda.empty_cache()
        gc.collect()
```

```
        return pred_mask.astype(np.uint8)

    def get_suspicious_mask(self, raw_mask: np.ndarray, smooth: bool = True)
        """
        Converts raw class mask to a binary 'Suspicious' mask.
        """
        suspicious = np.isin(raw_mask, list(self.suspicious_classes)).astype

        if not smooth:
            return suspicious

        # BALANCED SMOOTHING (5x5)
        # Big enough to remove salt-and-pepper noise.
        # Small enough to keep mines distinct from towns.
        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
        smoothed = cv2.morphologyEx(suspicious, cv2.MORPH_CLOSE, kernel)
        smoothed = cv2.morphologyEx(smoothed, cv2.MORPH_OPEN, kernel)

        return smoothed

class DetectionVisualizer:
    CLASS_COLORS = np.array([
        [0, 0, 0],       # 0: Background
        [255, 0, 0],     # 1: Building (Red)
        [255, 215, 0],   # 2: Road (Gold)
        [0, 0, 255],     # 3: Water (Blue)
        [160, 82, 45],   # 4: Barren (Brown)
        [34, 139, 34],   # 5: Forest (Green)
        [154, 205, 50]   # 6: Agri (LtGreen)
    ], dtype=np.uint8)

    def draw_segmentation_overlay(self, image: np.ndarray, mask: np.ndarray,
        safe_mask = np.clip(mask, 0, len(self.CLASS_COLORS) - 1)
        colored_mask = self.CLASS_COLORS[safe_mask]
        return (image * (1 - alpha) + colored_mask * alpha).astype(np.uint8)

    def draw_suspicious_overlay(self, image: np.ndarray, binary_mask: np.nda
        overlay = image.copy()
        overlay[binary_mask > 0] = [255, 0, 0] # Red highlight
        return cv2.addWeighted(image, 0.7, overlay, 0.3, 0)
```

Step 5: Two-Stage Detection

Stage 1: Overview Segmentation

1. Fetches a high-resolution overview image centered on the target coordinates using `SatelliteFetcher.fetch_overview()`.
2. Runs SegFormer segmentation to classify every pixel into land cover categories.
3. Generates a binary "suspicious" mask identifying regions with barren land, buildings, roads, or water.
4. Saves visualization outputs (raw overview, class overlay, suspicious overlay).

Stage 2: Grid Classification

1. Divides the overview image into an N×N grid (grid size based on overview radius / 2.5 km per cell).
2. For each grid cell:
   - Calculates the fraction of pixels flagged as suspicious
   - If suspicious ratio > 5%, retrieves high-resolution image centered on that cell
   - Runs the trained CNN classifier on the image
   - Gets result (mining detected or clean)
3. Draws detection results on the overview (red boxes for mining, green for clean suspicious areas, white grid lines). (in the vizualisation part)

Coordinate Conversion functions

`pixel_to_geo()` : Converts pixel coordinates (x, y) to geographic coordinates (lon, lat) given the image bounds.

`bbox_to_geo()` : Converts a pixel bounding box to geographic coordinates, returning both corner coordinates and centroid.

Outputs

- `OUTPUT_DIR/detections/overview_raw.jpg` : Original satellite overview
- `OUTPUT_DIR/detections/overview_segmentation_classes.jpg` : Color-coded land cover map
- `OUTPUT_DIR/detections/overview_segmentation_suspicious.jpg` : Binary suspicious regions
- `OUTPUT_DIR/detections/grid_analysis.jpg` : Final detection visualization with bounding boxes
- `OUTPUT_DIR/detections/classification_crops/` : Individual images for each classified cell
- Returns dictionary with detection results (coordinates, probabilities, labels)

In [ ]:
```
# ============================================================================
# STEP 5: Two-Stage Detection (Center + Radius)
# ============================================================================

def pixel_to_geo(pixel_coords: tuple, image_size: tuple, image_bounds: tuple
    """Convert pixel coordinates to geographic coordinates."""
    x, y = pixel_coords
    w, h = image_size
    lon_min, lat_min, lon_max, lat_max = image_bounds

    lon = lon_min + (x / w) * (lon_max - lon_min)
    lat = lat_max - (y / h) * (lat_max - lat_min)

    return (lon, lat)
```

```
def bbox_to_geo(bbox: tuple, image_size: tuple, image_bounds: tuple) -> dict
    """Convert pixel bounding box to geographic coordinates."""
    x1, y1, x2, y2 = bbox
    w, h = image_size
    lon_min, lat_min, lon_max, lat_max = image_bounds

    geo_lon_min = lon_min + (x1 / w) * (lon_max - lon_min)
    geo_lon_max = lon_min + (x2 / w) * (lon_max - lon_min)
    geo_lat_max = lat_max - (y1 / h) * (lat_max - lat_min)
    geo_lat_min = lat_max - (y2 / h) * (lat_max - lat_min)

    return {
        "lon_min": geo_lon_min, "lon_max": geo_lon_max,
        "lat_min": geo_lat_min, "lat_max": geo_lat_max,
        "centroid_lon": (geo_lon_min + geo_lon_max) / 2,
        "centroid_lat": (geo_lat_min + geo_lat_max) / 2
    }


def step5_detect_overview(
    model_dir: str,
    center_lat: Optional[float] = None,
    center_lon: Optional[float] = None,
    radius_km: float = OVERVIEW_RADIUS_KM,
    overview_max_dimension: int = 2048,
    classification_distance_km: float = 2.5,
    landsat_stats: Optional[Dict] = None,
    rate_limit_seconds: float = 0.5
) -> Dict:
    print("\n" + "=" * 60)
    print("STEP 5: Grid-Based Detection (Restored Classes)")
    print("=" * 60)

    model_path = Path(model_dir) / "best_model.pth"
    if not model_path.exists(): return {"error": "Classifier model not found

    # 1. Fetch Overview
    print("\n  [Stage 1] Fetching Overview...")
    fetcher = SatelliteFetcher(date_range=DATE_RANGE, max_cloud_cover=MAX_CL
    overview_image, overview_metadata = fetcher.fetch_overview(
        center_lat=center_lat, center_lon=center_lon,
        radius_km=radius_km, max_dimension=overview_max_dimension,
        target_stats=landsat_stats
    )
    if overview_image is None: return {"error": "Failed to fetch overview"}

    output_path = Path(OUTPUT_DIR) / "detections"
    output_path.mkdir(parents=True, exist_ok=True)
    Image.fromarray(overview_image).save(output_path / "overview_raw.jpg")

    # 2. Run Global Segmentation
    print("  Running Global Segmentation...")
    detector = MiningSegmentationDetector(model_name="wu-pr-gw/segformer-b2-
    visualizer = DetectionVisualizer()
```

```
    # Get Masks
    raw_mask = detector.predict_mask(overview_image)
    suspicious_mask = detector.get_suspicious_mask(raw_mask, smooth=True)

    # Save Visualizations
    color_overlay = visualizer.draw_segmentation_overlay(overview_image, raw
    Image.fromarray(color_overlay).save(output_path / "overview_segmentatior

    suspicious_overlay = visualizer.draw_suspicious_overlay(overview_image,
    Image.fromarray(suspicious_overlay).save(output_path / "overview_segment
    print(f"  Saved segmentations to {output_path}")

    # 3. Process Grid
    h, w = overview_image.shape[:2]
    total_width_km = radius_km * 2
    grid_n = max(2, int(total_width_km // 2.5))
    step_y, step_x = h // grid_n, w // grid_n

    print(f"\n  [Stage 2] Processing {grid_n}x{grid_n} Grid...")
    classifier = Predictor(model_path=str(model_path), backbone=BACKBONE, im
    results = []

    viz_img = Image.fromarray(overview_image).convert("RGBA")
    draw = ImageDraw.Draw(viz_img)
    bbox = center_to_bbox(center_lat, center_lon, radius_km)

    classification_dir = output_path / "classification_crops"
    classification_dir.mkdir(parents=True, exist_ok=True)

    for r in range(grid_n):
        for c in range(grid_n):
            y1, x1 = r * step_y, c * step_x
            y2 = h if r == grid_n - 1 else (r + 1) * step_y
            x2 = w if c == grid_n - 1 else (c + 1) * step_x

            # Analyze Mask
            cell_mask = suspicious_mask[y1:y2, x1:x2]
            suspicious_pixels = np.count_nonzero(cell_mask)
            suspicious_ratio = suspicious_pixels / cell_mask.size

            # Draw Grid
            draw.rectangle([x1, y1, x2, y2], outline=(255, 255, 255, 80), wi

            # DEBUG PRINT
            # print(f"    Grid ({r},{c}): Ratio {suspicious_ratio:.1%}")

            # Threshold > 5%
            if suspicious_ratio > 0.05:
                cy_px, cx_px = (y1 + y2) // 2, (x1 + x2) // 2
                cell_lat = bbox[3] - (cy_px / h) * (bbox[3] - bbox[1])
                cell_lon = bbox[0] + (cx_px / w) * (bbox[2] - bbox[0])

                rgb, _ = fetcher.fetch_image(
                    lat=cell_lat, lon=cell_lon,
                    distance_km=classification_distance_km,
```

```
                    target_stats=landsat_stats,
                    simulate_landsat=True
                )

                if rgb is not None:
                    # Save Crop
                    crop_filename = f"cell_{r}_{c}_{suspicious_ratio:.2f}.jp
                    crop_path = classification_dir / crop_filename
                    Image.fromarray(rgb).save(crop_path)

                    # Classify
                    pred = classifier.predict(str(crop_path))

                    if pred["is_mining"]:
                        draw.rectangle([x1, y1, x2, y2], outline=(255, 0, 0)
                        results.append({"lat": cell_lat, "lon": cell_lon, "p
                        print(f"    Cell ({r},{c}): 🚨 MINING ({pred['probab
                    else:
                        draw.rectangle([x1, y1, x2, y2], outline=(0, 255, 0)
                        print(f"    Cell ({r},{c}): Clean ({pred['probabilit
                else:
                    print(f"    Cell ({r},{c}): Fetch Failed")
            else:
                pass # Skip

    viz_img.convert("RGB").save(output_path / "grid_analysis.jpg")
    print(f"\n  Saved visualization to {output_path / 'grid_analysis.jpg'}")

    return {"results": results}
```

Step 6: Validation

This step was created to evaluate the trained model. **This part is quite badly done because we can't guarantee that the images we tested on were not part of the training test. This was because we had not enough data. To solve this problem, we would need the actual coordinates of the mining dams and make sure that they were not used in the model.)

Input Data

Because of the data we had, validation coordinates can be provided via:

1. `validation_csv` : Path to a CSV file with columns for latitude, longitude, and label ("mining" or "forest")
2. `known_coordinates` : List of dictionaries passed directly to the function

Evaluation

For each validation coordinate:

1. Fetch satellite imagery using the same parameters as training data collection
2. Save the image for inspection
3. Run the trained classifier to get a prediction
4. Compare prediction against ground-truth label
5. Update confusion matrix counters (TP, FP, TN, FN)

Metrics

- **Accuracy**: (TP + TN) / Total — overall correctness
- **Precision**: TP / (TP + FP) — of predicted positives, how many are correct
- **Recall**: TP / (TP + FN) — of actual positives, how many were detected
- **F1 Score**: Harmonic mean of precision and recall

For mining detection, recall is more important than precision (missing illegal mining is worse than a false alarm), but we are unsure of actual operational requirements.

Outputs

- `OUTPUT_DIR/validation/images/` : Fetched validation images
- `OUTPUT_DIR/validation/validation_results.json` : Per-sample results and aggregate metricsμ
- Returns dictionary with metrics

In [ ]:
```
# ============================================================================
# STEP 6: Validate Model Against Known Coordinates
# ============================================================================

def step6_validate(
    model_dir: str,
    validation_csv: Optional[str] = None,
    known_coordinates: Optional[List[Dict]] = None,
    sample_size: Optional[int] = None
) -> Dict:
    print("\n" + "=" * 60)
    print("STEP 6: Validating Model Against Known Coordinates")
    print("=" * 60)

    model_path = Path(model_dir) / "best_model.pth"

    if not model_path.exists():
        print(f"  Error: Model not found at {model_path}")
        return {"error": "Model not found"}

    coordinates = []

    if known_coordinates:
        coordinates = known_coordinates
    elif validation_csv and Path(validation_csv).exists():
        print(f"  Loading validation data from: {validation_csv}")
        with open(validation_csv, "r") as f:
```

```
            reader = csv.DictReader(f)
            for row in reader:
                try:
                    lat_col = next((col for col in row.keys() if col.lower()
                    lon_col = next((col for col in row.keys() if col.lower()
                    label_col = next((col for col in row.keys() if col.lower

                    if lat_col and lon_col and label_col:
                        coordinates.append({
                            "lat": float(row[lat_col]),
                            "lon": float(row[lon_col]),
                            "label": row[label_col].lower()
                        })
                except (ValueError, KeyError):
                    continue

    if not coordinates:
        print("  No validation coordinates provided. Skipping validation.")
        return {"skipped": True}

    if sample_size and len(coordinates) > sample_size:
        import random
        coordinates = random.sample(coordinates, sample_size)

    print(f"  Validating on {len(coordinates)} samples")

    output_path = Path(OUTPUT_DIR) / "validation"
    output_path.mkdir(parents=True, exist_ok=True)
    images_dir = output_path / "images"
    images_dir.mkdir(parents=True, exist_ok=True)

    fetcher = SatelliteFetcher(date_range=DATE_RANGE, max_cloud_cover=MAX_CL
    classifier = Predictor(model_path=str(model_path), backbone=BACKBONE, im

    results = []
    tp, fp, tn, fn = 0, 0, 0, 0

    for i, coord in enumerate(coordinates):
        rgb, metadata = fetcher.fetch_image(lat=coord["lat"], lon=coord["lor

        if rgb is None:
            results.append({"lat": coord["lat"], "lon": coord["lon"], "knowr
            continue

        img_path = images_dir / f"val_{i:04d}_{coord['label']}.jpg"
        Image.fromarray(rgb).save(img_path, quality=95)

        # Free rgb after saving
        del rgb

        pred = classifier.predict(str(img_path))
        predicted_label = "mining" if pred["is_mining"] else "forest"
        known_is_mining = coord["label"] in ["mining", "mine", "positive", "

        correct = (pred["is_mining"] == known_is_mining)
```

```
        if known_is_mining and pred["is_mining"]: tp += 1
        elif known_is_mining and not pred["is_mining"]: fn += 1
        elif not known_is_mining and pred["is_mining"]: fp += 1
        else: tn += 1

        results.append({"lat": coord["lat"], "lon": coord["lon"], "known_lab

        if (i + 1) % 10 == 0:
            print(f"    Processed {i + 1}/{len(coordinates)}")
            gc.collect()

        time.sleep(0.5)

    total = tp + tn + fp + fn
    metrics = {
        "total": len(coordinates), "processed": total,
        "accuracy": (tp + tn) / total if total > 0 else 0,
        "precision": tp / (tp + fp) if (tp + fp) > 0 else 0,
        "recall": tp / (tp + fn) if (tp + fn) > 0 else 0,
        "f1": 2 * (tp / (tp + fp)) * (tp / (tp + fn)) / ((tp / (tp + fp)) +
        "true_positives": tp, "false_positives": fp, "true_negatives": tn, "
    }

    print(f"\n  VALIDATION RESULTS")
    print(f"  Accuracy: {metrics['accuracy']:.1%}, Precision: {metrics['prec

    output = {"metrics": metrics, "results": results}

    results_path = output_path / "validation_results.json"
    with open(results_path, "w") as f:
        json.dump(output, f, indent=2)

    return output
```

`Main()`

Starts the whole code. Takes parser arguments explained below

Pipeline Modes

This needed to be implemented because it makes 0 sese to re-execute the whole code every time. That is why these were implemented and our whole pipeline was designed so that the steps were independant:

Full Pipeline (`skip_training=False, skip_data_collection=False`): Runs all six steps sequentially. To be used on iinitial setup or when retraining on new data.

Inference Only (`skip_training=True`): Skips Steps 1–4 and uses an existing trained model. Requires `OUTPUT_DIR/model/best_model.pth` to exist.

Retrain Without Data Collection (`skip_data_collection=True`): Skips Steps 1–2 but rebuilds the dataset and retrains. this was to test different model augmentations and model settings. Allows us to do without re-fetching the images.

Function Parameters

- `center_lat` , `center_lon` : Target coordinates for Step 5 detection (optional; if not provided, Step 5 is skipped)
- `radius_km` : Scan radius around center point
- `overview_max_dimension` : Maximum pixel dimension for overview image
- `classification_distance_km` : Size of individual classification patches
- `validation_csv` , `known_coordinates` : Ground-truth data for Step 6

CLI

when this was coded, it was initially built to be used from the terminal (best practive implementation) here are the params:

- `--skip-training` : Use existing model
- `--skip-data` : Skip data collection
- `--center LAT,LON` : Target coordinates
- `--radius KM` : Scan radius
- `--overview-size PX` : Maximum overview dimension
- `--classification-distance KM` : Classification patch size
- `--validate PATH` : Custom validation CSV path

usage (Cf README):

```
python main.py --skip-training --center="-14.2,-49.4" --radius 10
```

In [ ]:
```
def main(
    skip_data_collection: bool = False,
    skip_training: bool = False,
    center_lat: Optional[float] = None,
    center_lon: Optional[float] = None,
    radius_km: float = OVERVIEW_RADIUS_KM,
    overview_max_dimension: int = 2048,
    classification_distance_km: float = 2.5,
    validation_csv: Optional[str] = None,
    known_coordinates: Optional[List[Dict]] = None
):
    print("\n" + "=" * 60)
    print("ILLEGAL MINING DETECTION PIPELINE")
    print("=" * 60)
    print(f"\nOutput directory: {OUTPUT_DIR}")

    outputs = {}
    model_dir = Path(OUTPUT_DIR) / "model"
    landsat_stats = None
```

```
if not skip_training:
    if not skip_data_collection:
        outputs["step1"] = step1_collect_mines()
        landsat_stats = outputs["step1"]["landsat_stats"]
        outputs["step2"] = step2_collect_forest(landsat_stats=landsat_st
        outputs["step3"] = step3_build_dataset(
            mines_dir=outputs["step1"]["output_dir"],
            forest_dir=outputs["step2"]["output_dir"],
            landsat_stats=landsat_stats
        )
    else:
        print("Skipping data collection (steps 1-2)")
        outputs["step3"] = {"output_dir": str(Path(OUTPUT_DIR) / "datase
        stats_file = Path(OUTPUT_DIR) / "landsat_stats.json"
        if stats_file.exists():
            with open(stats_file, "r") as f:
                landsat_stats = json.load(f)

    outputs["step4"] = step4_train(dataset_dir=outputs["step3"]["output_
    model_dir = outputs["step4"]["model_dir"]
else:
    print("Skipping training (steps 1-4), using existing model")
    if not (Path(model_dir) / "best_model.pth").exists():
        print(f"  ERROR: No model found at {model_dir}/best_model.pth")
        return outputs

    stats_file = Path(OUTPUT_DIR) / "landsat_stats.json"
    if stats_file.exists():
        with open(stats_file, "r") as f:
            landsat_stats = json.load(f)

outputs["step5"] = step5_detect_overview(
    model_dir=str(model_dir),
    center_lat=center_lat,
    center_lon=center_lon,
    radius_km=radius_km,
    overview_max_dimension=overview_max_dimension,
    classification_distance_km=classification_distance_km,
    landsat_stats=landsat_stats
)

outputs["step6"] = step6_validate(
    model_dir=str(model_dir),
    validation_csv=validation_csv or VALIDATION_CSV,
    known_coordinates=known_coordinates,
)

print("\n" + "=" * 60)
print("PIPELINE COMPLETE")
print("=" * 60)

if outputs.get("step5") and not outputs["step5"].get("skipped"):
    n_mining = outputs["step5"].get("n_confirmed_mining", 0)
    print(f"  Detections: {n_mining} mining zones detected")
```

```
    return outputs


if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Illegal Mining Detection P
    parser.add_argument("--skip-training", action="store_true", help="Skip t
    parser.add_argument("--skip-data", action="store_true", help="Skip data
    parser.add_argument("--center", type=str, help="Center coordinates: LAT,
    parser.add_argument("--radius", type=float, default=OVERVIEW_RADIUS_KM,
    parser.add_argument("--overview-size", type=int, default=2048, help="Max
    parser.add_argument("--classification-distance", type=float, default=2.5
    parser.add_argument("--validate", type=str, help="Path to validation CSV

    args = parser.parse_args()

    center_lat, center_lon = None, None
    if args.center:
        parts = args.center.split(",")
        if len(parts) == 2:
            center_lat = float(parts[0])
            center_lon = float(parts[1])
        else:
            print("Error: --center must be LAT,LON (e.g., -14.2,-49.4)")
            exit(1)


    # python3 main.py --skip-training --center="-14.2,-49.4" --radius 10 --c
    main(
        skip_training=args.skip_training,
        skip_data_collection=args.skip_data,
        center_lat=center_lat,
        center_lon=center_lon,
        radius_km=args.radius,
        overview_max_dimension=args.overview_size,
        classification_distance_km=args.classification_distance,
        validation_csv=args.validate
    )
```

In [ ]:
```
# %python3 main.py --skip-training --center="-14.2,-49.4" --radius 10 --over
```