SVAMITVA
Feature Extraction System

AI-Powered Drone Imagery Analysis
for Rural Property Mapping

Digital University Kerala (DUK)

AI Model developed by DUK Students for Feature Extraction

Date: February 13, 2026

Deep Learning | *Semantic Segmentation* | *DeepLabV3+* | *EfficientNet-B4*
PyTorch | *Streamlit* | *OpenCV* | *GIS Integration*

Table of Contents

1. Abstract / Executive Summary

The **SVAMITVA (Survey of Villages Abadi and Mapping with Improvised Technology in Village Areas)** scheme is a flagship initiative by the Government of India, launched in 2020, aimed at providing property cards to rural households through the mapping of land parcels using drone technology. The scheme addresses a fundamental gap in rural governance: the absence of accurate, digitized property records for village abadi (inhabited) areas, which has long hindered financial inclusion, land dispute resolution, and rural planning.

The Problem: Currently, the process of extracting features from high-resolution drone imagery is predominantly manual. Trained GIS operators spend hours delineating building footprints, roads, waterbodies, and infrastructure elements from orthomosaic images. This manual approach is not only time-consuming and expensive but also prone to human error and inconsistency. With the SVAMITVA scheme targeting over 6.62 lakh villages across India, the scale of the task demands an automated solution.

Our Solution: We developed an AI-powered feature extraction system that uses deep learning-based semantic segmentation to automatically identify and classify features from drone imagery. Our system employs a **DeepLabV3+ architecture with an EfficientNet-B4 backbone**, trained on drone imagery to classify pixels into 10 distinct categories including building footprints with roof type classification (RCC, Tiled, Tin, Others), roads, waterbodies, and infrastructure elements (transformers, tanks, wells). The system processes raw drone images and outputs georeferenced shapefiles ready for integration with existing GIS workflows used by the Survey of India.

Our pipeline achieves a mean Intersection over Union (IoU) of **0.381** and pixel accuracy of **69.5%** after training on just 20 drone images with auto-generated labels. While these numbers have significant room for improvement, they demonstrate the viability of the approach, especially considering the limitations of our training data. The system includes a user-friendly Streamlit web interface that allows operators to upload images, select feature classes, adjust post-processing parameters, and export results as shapefiles.

2. Problem Statement

2.1 Background: The SVAMITVA Scheme

The SVAMITVA scheme was launched by the Hon'ble Prime Minister on April 24, 2020 (National Panchayati Raj Day) as a central sector scheme. The scheme uses drone technology (Unmanned Aerial Vehicles) to create high-resolution maps of rural inhabited areas. These maps serve as the basis for issuing 'Property Cards' (also known as 'Sampatti Patrak' or 'Title Deed') to rural property owners, thereby providing them with a legal document of ownership.

The key objectives of SVAMITVA include: (a) creating a record of rights for rural households, (b) enabling property-based financial services and credit access, (c) reducing property-related disputes in rural areas, (d) supporting comprehensive village-level planning, and (e) creating an accurate land records system for gram panchayats to assess and collect property taxes.

2.2 The Feature Extraction Challenge

The drone survey process generates high-resolution orthomosaic images (typically 2-5 cm Ground Sampling Distance) of village areas. From these images, GIS analysts must extract several categories of features to prepare the village property maps. The current manual process involves:

- **Building Footprint Delineation:** Manually tracing the outline of every building structure, which can number in the hundreds for a single village.
- **Roof Type Classification:** Identifying the roof material (RCC/concrete, clay tiles, tin/metal sheets, or other materials) for each building, which is relevant for property valuation.
- **Road Network Mapping:** Tracing paved and unpaved roads, lanes, and pathways throughout the village.
- **Waterbody Identification:** Marking ponds, tanks, streams, and other water features.
- **Infrastructure Mapping:** Locating transformers, water tanks, wells, and other public infrastructure.

A single village map can take a trained operator **4-8 hours** to complete manually. Given that the SVAMITVA scheme aims to cover over 6 lakh villages, the total manual effort required would be astronomical. Our automated system aims to reduce this time to **minutes per village**, with human operators only needed for quality assurance and edge case correction.

2.3 Target Feature Classes

Class ID	Class Name	Description	Priority
0	Background	Vegetation, bare ground, shadows	Low
1	Building_RCC	Reinforced Cement Concrete roofs	High
2	Building_Tiled	Clay/ceramic tile roofs	High
3	Building_Tin	Tin/metal sheet roofs	High

4	Building_Other	Other roof materials	High
5	Road	Paved and unpaved roads	High
6	Waterbody	Ponds, tanks, streams	Medium
7	Transformer	Electrical transformers	Medium
8	Tank	Water storage tanks	Medium
9	Well	Open and bore wells	Medium

Table 1: Feature classes for the SVAMITVA segmentation model

3. Our Approach & Methodology

3.1 Why Semantic Segmentation?

We evaluated two main approaches for feature extraction from drone imagery: **object detection** (e.g., YOLO, Faster R-CNN) and **semantic segmentation** (e.g., U-Net, DeepLabV3+). While object detection excels at locating and classifying discrete objects with bounding boxes, it falls short for our use case for several reasons:

• **Pixel-level precision:** Property mapping requires exact building footprint boundaries, not just bounding boxes. Semantic segmentation provides per-pixel classification, which directly translates to polygon boundaries for shapefiles.

• **Irregular shapes:** Buildings in rural India come in highly irregular shapes. Bounding boxes would include significant background area, making area calculations inaccurate.

• **Continuous features:** Roads and waterbodies are continuous, elongated features that cannot be meaningfully represented by bounding boxes.

• **Multi-class dense prediction:** We need to classify every pixel in the image, not just detect objects. Background, roads, and buildings all need to be labeled simultaneously.

3.2 Model Architecture Selection

We experimented with several segmentation architectures before settling on our final choice:

Phase 1 - U-Net with ResNet50: Our initial attempt used U-Net with a ResNet50 backbone. While U-Net is excellent for biomedical segmentation, we found it struggled with the multi-scale nature of our task. Buildings range from tiny huts (50x50 pixels) to large community halls (500x500 pixels), and U-Net's fixed receptive field couldn't capture this range effectively.

Phase 2 - FPN with ResNet50: We briefly tried Feature Pyramid Networks (FPN), which handle multi-scale features better than vanilla U-Net. Performance improved slightly (+1.2% mIoU) but was still not satisfactory, particularly for small infrastructure objects.

Phase 3 - DeepLabV3+ with EfficientNet-B4: Our final architecture combines DeepLabV3+'s Atrous Spatial Pyramid Pooling (ASPP) module for multi-scale feature extraction with EfficientNet-B4's efficient and powerful feature encoding. This combination gave us approximately **3% higher mIoU** compared to the ResNet50-based approaches while using fewer parameters.

3.3 Why EfficientNet-B4 as Backbone?

EfficientNet uses a compound scaling method that uniformly scales network depth, width, and input resolution using a set of fixed scaling coefficients. EfficientNet-B4 specifically provides:

• **Parameter Efficiency:** ~19M parameters compared to ResNet50's ~25M, yet achieves higher accuracy on ImageNet (82.9% vs 76.1% top-1 accuracy).

• **Better Feature Extraction:** The compound scaling ensures that the network captures features at multiple levels of abstraction, which is crucial for distinguishing between roof types.

• **ImageNet Pre-training:** Using ImageNet-pretrained weights as initialization dramatically reduces the amount of domain-specific data needed, which is critical given our limited dataset of 20 images.

• **Mobile-Friendly:** The efficient architecture means faster inference, which matters for real-time processing of drone imagery in field conditions.

4. Technical Architecture (Deep Dive)

4.1 System Architecture Overview

Our system follows a modular pipeline architecture designed for flexibility and maintainability. The end-to-end flow can be summarized as:

Stage	Component	Description
Input	Image Loader	Supports JPEG, PNG, TIFF, GeoTIFF formats
Preprocessing	Augmentation	Resize, normalize, ImageNet stats
Model	DeepLabV3+	EfficientNet-B4 encoder + ASPP + decoder
TTA	Flip Averaging	Horizontal + Vertical flip ensembling
Post-Processing	Morphology	Opening, closing, hole filling, smoothing
Vectorization	Polygonization	Raster-to-vector with Douglas-Peucker
Output	Shapefiles	ESRI Shapefile format with attributes

Table 2: System pipeline stages

4.2 DeepLabV3+ Architecture

DeepLabV3+ (Chen et al., 2018) is a state-of-the-art semantic segmentation architecture that combines the strengths of encoder-decoder networks with atrous (dilated) convolutions. The key innovation is the **Atrous Spatial Pyramid Pooling (ASPP)** module, which applies multiple parallel atrous convolutions at different dilation rates to capture multi-scale context.

The ASPP module in our configuration applies atrous convolutions with dilation rates of 6, 12, and 18, along with a 1x1 convolution and image-level global average pooling. This allows the model to simultaneously analyze features at multiple spatial scales — capturing both the fine details of individual building edges and the broader spatial context of village layout patterns.

The decoder module takes the encoder's low-level features and the ASPP output, upsamples them, and concatenates them to produce the final segmentation map. This skip connection from the encoder to the decoder helps preserve fine spatial details that would otherwise be lost during downsampling.

4.3 EfficientNet-B4 Encoder

EfficientNet (Tan & Le, 2019) introduced a principled approach to scaling neural networks using a compound coefficient ϕ that uniformly scales depth (d = α^ϕ), width (w = β^ϕ), and resolution (r = γ^ϕ). For EfficientNet-B4 specifically:

Parameter	Value
Input Resolution	380 x 380
Depth Coefficient	1.8
Width Coefficient	1.4

Parameters	~19.3M
Top-1 Accuracy (ImageNet)	82.9%
MBConv Blocks	7 stages with squeeze-and-excitation

Table 3: EfficientNet-B4 specifications

4.4 Model Definition (Code)

We use the `segmentation_models_pytorch` library for constructing the model, which provides a clean API for combining different encoders with segmentation heads:

```
class SVAMITVASegmentationModel(nn.Module):
    def __init__(self, num_classes=10, encoder="efficientnet-b4",
                 encoder_weights="imagenet", activation=None):
        super().__init__()
        self.num_classes = num_classes
        self.model = smp.DeepLabV3Plus(
            encoder_name=encoder,
            encoder_weights=encoder_weights,
            in_channels=3,
            classes=num_classes,
            activation=activation,
        )

    def forward(self, x):
        return self.model(x)

    def predict_proba(self, x):
        return torch.softmax(self.forward(x), dim=1)
```

5. Training Pipeline

5.1 Data Preparation: Auto-Labeling

One of the biggest challenges we faced was the lack of annotated training data. Manually labeling drone images at the pixel level is extremely time-consuming. To bootstrap our training process, we developed an **auto-labeling pipeline** that uses color and texture heuristics in the HSV (Hue, Saturation, Value) color space to generate approximate segmentation masks.

Our HSV-based classification works as follows: Roads are identified by low saturation ($S < 40$) and medium brightness (V: 80-200), as they tend to be grayish. Tiled roofs show a distinctive orange-red hue (H: 5-25) with high saturation. RCC roofs appear as dark gray regions (low S, low V). Vegetation (background) is identified by green hues (H: 30-85). After pixel-level classification, we apply morphological refinement including closing operations (to fill gaps), opening operations (to remove noise), and contour-based filtering with minimum area thresholds.

We also apply convex hull approximation to building contours, which makes them look more like actual building footprints instead of blobby shapes. This auto-labeling process generated masks for 16 training images and 4 validation images — a small but sufficient dataset to begin training.

5.2 Augmentation Pipeline

Given our extremely small dataset (20 images total), heavy data augmentation was essential to prevent overfitting. We used the **Albumentations** library for its speed and its ability to jointly transform images and masks. Our augmentation pipeline includes:

Augmentation	Parameters	Probability
Horizontal Flip	-	0.5
Vertical Flip	-	0.5
Random Rotate 90	-	0.5
Shift-Scale-Rotate	shift=0.1, scale=0.2, rot=45°	0.5
Elastic Transform	α=120, σ=6.0	0.3
Grid Distortion	default params	0.2
Optical Distortion	default params	0.2
CLAHE	default params	0.3
Brightness/Contrast	±0.2	0.5
Hue-Sat-Value	hue=10, sat=20, val=10	0.3
Gaussian Blur/Noise	blur 3-7, var 10-50	0.2
Coarse Dropout	8 holes, 32x32 max	0.3

Table 4: Data augmentation pipeline

We also resize images to 576x576 first and then apply a random crop to 512x512 during training, which provides slight scale variation without an explicit scale augmentation step.

Validation images are simply resized to the target resolution without any augmentation. All images are normalized using ImageNet statistics (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) since our encoder is pre-trained on ImageNet.

5.3 Loss Function: Focal + Dice Combined Loss

Class imbalance is a major challenge in our dataset — over 80% of pixels in a typical drone image are background (vegetation, bare ground). Using standard cross-entropy loss, the model quickly learns to predict everything as background and achieves ~80% accuracy while being completely useless.

Our solution is a **combined Focal + Dice loss** with configurable weights:

Focal Loss (Lin et al., 2017) adds a modulating factor (1 - p_t)^γ to the standard cross-entropy loss, which downweights easy examples (like background pixels the model is already confident about) and focuses training on hard examples (building edges, ambiguous pixels). This naturally handles class imbalance without requiring explicit oversampling.

Dice Loss directly optimizes the Dice coefficient (equivalent to F1 score), which measures the overlap between predicted and ground truth masks. Unlike cross-entropy, Dice loss treats the entire mask holistically, making it robust to class imbalance since it normalizes by the size of each class.

```
class FocalDiceLoss(nn.Module):
    def __init__(self, focal_weight=0.4, dice_weight=0.6,
                 class_weights=None):
        super().__init__()
        self.focal_weight = focal_weight
        self.dice_weight = dice_weight
        self.focal_loss = smp.losses.FocalLoss(mode="multiclass")
        self.dice_loss = smp.losses.DiceLoss(
            mode="multiclass", classes=list(range(10))
        )
        self.class_weights = class_weights

    def forward(self, predictions, targets):
        focal = self.focal_loss(predictions, targets)
        dice = self.dice_loss(predictions, targets)
        total = self.focal_weight * focal + self.dice_weight * dice
        if self.class_weights is not None:
            ce = F.cross_entropy(predictions, targets,
                                 weight=self.class_weights)
            total = total + 0.1 * ce
        return total
```

5.4 Class Weights Strategy

In addition to the Focal + Dice loss, we apply per-class weights to the auxiliary cross-entropy term. These weights were determined empirically based on the inverse frequency of each class in our training set:

Class	Weight	Rationale
Background	0.3	Dominant class (~80%), heavily downweighted
Building_RCC	2.5	Important target class, moderately rare
Building_Tiled	2.5	Important target class, moderately rare

Building_Tin	2.5	Important but challenging to detect
Building_Other	2.5	Catch-all building category
Road	1.5	Common but elongated, needs moderate weight
Waterbody	2.0	Moderate rarity in typical villages
Transformer	4.0	Very rare, tiny objects, needs high weight
Tank	4.0	Very rare, needs high weight
Well	4.0	Very rare, needs high weight

Table 5: Per-class weights for loss function

5.5 Training Configuration & Details

We maintained two training configurations — one optimized for GPU training and a lighter version for CPU-only environments (which was unfortunately our primary setup during the hackathon).

Parameter	GPU Config	CPU Config
Input Size	512 x 512	256 x 256
Batch Size	4	2
Effective Batch Size	8 (grad accum 2)	2
Learning Rate	3e-4	1e-3
Weight Decay	1e-4	1e-4
Num Workers	4	0
Max Epochs	150	30
Warmup Epochs	5	3
Scheduler	Cosine Annealing	Cosine Annealing
Min LR	1e-7	1e-7
Patience (Early Stop)	20	10
Gradient Clipping	max_norm=1.0	max_norm=1.0
Mixed Precision	Yes (AMP)	No

Table 6: Training configurations for GPU and CPU environments

Key training techniques we employed include: **Gradient Accumulation** (simulates batch size of 8 by accumulating gradients over 2 mini-batches before updating weights), **Learning Rate Warmup** (linearly ramps up LR from 0 to target over the first 5 epochs to prevent unstable training with pre-trained weights), **Cosine Annealing** (smoothly decays LR following a cosine curve for better convergence), and **Gradient Clipping** (clips gradient norms to 1.0 to prevent exploding gradients).

5.6 Training Results

Due to computational constraints, we trained the model for **14 epochs** on CPU with the 256x256 configuration. Our best results were:

- **Best Mean IoU:** 0.381 (achieved at epoch 12)
- **Best Pixel Accuracy:** 69.5%
- **Best Mean Dice:** 0.467
- **Training Loss:** Converged from ~2.1 to ~0.85
- **Validation Loss:** Converged from ~1.9 to ~1.1

6. Inference Pipeline

6.1 Test-Time Augmentation (TTA)

Test-Time Augmentation is a technique where we run inference on multiple augmented versions of the same image and average the predictions. This reduces prediction noise and improves accuracy by approximately **1-2% mIoU** with minimal computational overhead. Our TTA strategy uses:

- **Original image:** Standard forward pass
- **Horizontal flip:** Flip input, predict, flip prediction back
- **Vertical flip:** Flip input, predict, flip prediction back

The probability maps from all three passes are averaged element-wise, and the final class prediction is taken as the argmax of the averaged probabilities. This ensemble approach smooths out prediction inconsistencies, especially at object boundaries.

6.2 Sliding Window for Large Images

Drone images are typically much larger than the model's input resolution (e.g., 4000x3000 pixels vs. our 512x512 training size). Resizing the entire image to 512x512 would lose critical spatial detail. Instead, we use a **sliding window approach**:

We slide a 512x512 window across the image with a stride of 384 pixels (i.e., 128 pixels of overlap). For each window, we run the model and accumulate the probability maps. Where windows overlap, the probabilities are averaged, which creates smooth transitions between windows and avoids the harsh edge artifacts that would occur with non-overlapping tiles. We also ensure that the windows cover the image edges by adding extra windows if the last window doesn't reach the image boundary.

6.3 Confidence Thresholding & Class Masking

After generating the probability map, we apply two additional filtering steps. **Confidence Thresholding:** Pixels where the maximum class probability is below 0.3 (30%) are assigned to background (class 0). This prevents the model from making low-confidence predictions that would appear as noise in the output. **Class Masking:** Since we haven't trained on all 10 classes (classes 3, 6, 7, 8, 9 have insufficient training data), we mask out untrained classes by setting their logits to $-\infty$ before the argmax operation. This ensures the model only predicts classes it has actually learned to recognize: Background, Building_RCC, Building_Tiled, Building_Other, and Road.

```
def _apply_class_masking(self, probs):
    for c in self._invalid_classes:
        probs[c] = -1e9  # effectively zero after softmax
    mask = np.argmax(probs, axis=0)
    max_probs = np.max(probs, axis=0)
    mask[max_probs < self.CONFIDENCE_THRESHOLD] = 0
    return mask, probs
```

7. Post-Processing Pipeline

Raw model outputs contain significant noise — isolated pixels, jagged edges, small holes inside buildings, and fragmented road segments. Our post-processing pipeline cleans up these artifacts to produce professional-quality outputs suitable for GIS workflows.

7.1 Morphological Operations

We apply two morphological operations using elliptical structuring elements (3x3 kernel):

- **Opening (erosion + dilation):** Removes small isolated clusters of pixels that are likely noise. For example, a single pixel classified as 'building' surrounded by background is removed.
- **Closing (dilation + erosion):** Fills small gaps and holes within detected objects. This is particularly useful for buildings where the model may miss a few pixels inside the footprint.

We use elliptical kernels rather than rectangular ones because they produce smoother, more natural results — building edges in real life are rarely perfectly square.

7.2 Small Object Removal

After morphological operations, we use **connected component analysis** (8-connectivity) to identify separate objects and remove those below a minimum area threshold. Different classes use different thresholds: buildings require at least 50 pixels, roads 100 pixels, waterbodies 200 pixels, and infrastructure elements 20 pixels (since they are inherently small objects like transformers).

7.3 Hole Filling

Building footprints sometimes have internal holes where the model failed to classify interior pixels correctly (e.g., a courtyard or shadow area within a building). We use contour-based hole filling: find contours of the inverted binary mask (which correspond to holes), and fill them in. This ensures that building polygons are solid without internal voids.

7.4 Boundary Smoothing

The final step applies Gaussian blur (5x5 kernel) followed by re-thresholding at 127 to smooth jagged pixel-level boundaries into cleaner curves. We apply this for 2 iterations, which we found to be the sweet spot between smoothness and preserving geometric accuracy.

7.5 Polygon Simplification (Douglas-Peucker)

When converting raster masks to vector polygons for shapefile export, we apply the **Douglas-Peucker algorithm** with a configurable tolerance (default: 1.0). This algorithm reduces the number of vertices in each polygon while preserving the overall shape within the specified tolerance. This is critical for generating manageable shapefiles — without simplification, a single building polygon might have thousands of vertices (one per pixel on the boundary), making the shapefile extremely large and slow to render in GIS software.

8. Code Architecture

8.1 Project File Structure

File	Description
`app.py`	Streamlit web application (main entry point for users)
`src/config.py`	All hyperparameters, class definitions, paths, and configs
`src/model.py`	Model definition (DeepLabV3+) and loss function (Focal+Dice)
`src/dataset.py`	PyTorch Dataset class and augmentation pipelines
`src/train.py`	Full training loop with warmup, scheduling, checkpointing
`src/inference.py`	Inference pipeline with TTA and sliding window
`src/postprocess.py`	Morphological post-processing of segmentation masks
`src/vectorize.py`	Raster-to-vector conversion and shapefile generation
`src/metrics.py`	IoU, Dice, Precision, Recall, F1, Accuracy calculations
`src/utils.py`	Helper utilities (logger, device detection, GeoTIFF loading)
`src/auto_label.py`	HSV-based auto-labeling for bootstrapping training data

Table 7: Project file structure

8.2 Module Interaction Flow

The modules interact in a clear dependency chain. `config.py` serves as the central configuration hub that all other modules import from. `model.py` defines the neural network architecture and loss function. `dataset.py` handles data loading and augmentation. `train.py` orchestrates the training loop, pulling together the model, dataset, and metrics modules. `inference.py` loads a trained checkpoint and runs predictions. `postprocess.py` cleans up the raw predictions, and `vectorize.py` converts them to shapefiles. Finally, `app.py` ties everything together in a user-friendly web interface.

The flow for a typical inference request is: **config** → **model** (load checkpoint) → **inference** (predict with TTA) → **postprocess** (clean mask) → **vectorize** (generate shapefiles) → **app** (display results).

8.3 Key Code: Training Loop

```
def train_epoch(self, train_loader, epoch):
    self.model.train()
    self.optimizer.zero_grad()
    for batch_idx, batch in enumerate(train_loader):
        images = batch["image"].to(self.device)
        masks = batch["mask"].to(self.device)
        outputs = self.model(images)
        loss = self.criterion(outputs, masks)
        loss = loss / self.accumulation_steps
        loss.backward()

        if (batch_idx + 1) % self.accumulation_steps == 0:
```

```
torch.nn.utils.clip_grad_norm_(
    self.model.parameters(), self.max_norm
)
self.optimizer.step()
self.optimizer.zero_grad()
```

9. Web Application (Streamlit)

We built the user-facing application using **Streamlit**, which allowed us to rapidly prototype a professional-looking interface during the hackathon without spending time on frontend development. Streamlit's reactive programming model means the app automatically updates when users interact with controls.

9.1 Features

- **Image Upload:** Drag-and-drop interface supporting JPEG, PNG, TIFF, and GeoTIFF formats. GeoTIFF files preserve geospatial metadata for accurate coordinate transformation in shapefiles.
- **Class Selection:** Sidebar checkboxes allow users to enable/disable specific feature classes. Classes without sufficient training data are marked with a warning icon.
- **Post-Processing Controls:** Toggle post-processing on/off and adjust polygon simplification tolerance with a slider (0.0 to 5.0).
- **Pixel Size Configuration:** Users can specify the ground sampling distance (in meters) for accurate area calculations.
- **Real-Time Visualization:** The original image and colored prediction mask are displayed side-by-side for easy comparison.
- **Statistics Dashboard:** Automatically calculates per-class object counts and areas, displayed both as a table and an interactive Plotly bar chart.
- **Export Options:** Download the raw mask (PNG), colored visualization (PNG), or georeferenced shapefiles (ZIP containing .shp, .shx, .dbf, .prj files).

9.2 Model Caching

Loading the model takes several seconds due to the size of EfficientNet-B4. We use Streamlit's `@st.cache_resource` decorator to cache the loaded model in memory, so it only loads once per session. We pass the selected valid classes as a tuple (since Streamlit can't hash lists) to ensure the model reloads if the user changes class selection.

9.3 Color Legend

The application displays an interactive color legend built with Plotly showing the mapping between colors and feature classes. Each class has a carefully chosen color: red shades for buildings, brown for roads, blue for water, and distinct colors for infrastructure elements. Background is rendered as black and excluded from the legend for clarity.

10. Results & Analysis

10.1 Training Performance

The model was trained for 14 epochs on CPU using the 256x256 configuration. Training was limited by computational resources — each epoch took approximately 25 minutes on a quad-core CPU. The training curves show steady improvement in both loss and IoU metrics:

Metric	Epoch 1	Epoch 7	Epoch 12 (Best)	Epoch 14
Train Loss	2.13	1.24	0.91	0.85
Val Loss	1.95	1.42	1.08	1.12
Mean IoU	0.089	0.241	0.381	0.372
Accuracy	34.2%	55.8%	69.5%	68.7%
Mean Dice	0.142	0.318	0.467	0.453

Table 8: Training metrics across epochs (best epoch highlighted)

10.2 Per-Class Analysis

Performance varied significantly across classes. **Buildings (especially RCC and Tiled)** and **Roads** showed the best detection results, likely because they have the most distinctive visual features and constitute the majority of non-background pixels in our training data.

Building_RCC (class 1) achieved the highest per-class IoU of approximately 0.45, benefiting from the distinctive dark gray appearance of concrete roofs. Building_Tiled (class 2) achieved ~0.38 IoU thanks to the easily recognizable orange-brown color of clay tiles. Roads (class 5) achieved ~0.35 IoU but suffered from fragmentation in areas with tree canopy cover. Infrastructure classes (7-9) were not evaluated as they lacked sufficient training data.

10.3 Limitations

- **Auto-generated labels:** The HSV-based auto-labeling introduces systematic errors. For example, dark shadows are sometimes classified as RCC roofs, and certain soil colors can be confused with tiled roofs. This noise in the training labels puts a ceiling on achievable accuracy.
- **Limited training data:** Only 20 drone images (16 train, 4 validation) were used. Deep learning models typically require hundreds or thousands of annotated images for robust performance.
- **Low resolution training:** Due to CPU constraints, we trained at 256x256, which loses fine spatial details. Training at 512x512 or higher on a GPU would significantly improve results.
- **Class imbalance:** Despite our weighted loss strategy, the model still shows bias toward predicting background for ambiguous pixels.
- **Limited geographic diversity:** All training images are from a single region, which means the model may not generalize well to villages with different architectural styles or landscapes.

11. Tools & Technologies Used

Category	Technology	Purpose
Language	Python 3.11	Primary development language
Deep Learning	PyTorch 2.x	Neural network framework
Segmentation	segmentation-models-pytorch	Pre-built architectures (DeepLabV3+)
Augmentation	Albumentations	Fast image and mask augmentation
Image Processing	OpenCV (headless)	Image I/O, morphological operations
Image Processing	Pillow (PIL)	Image format conversion
Web Framework	Streamlit	Rapid prototyping of web interface
Visualization	Plotly	Interactive charts and legends
Visualization	Matplotlib/Seaborn	Training curve plots
GIS	Rasterio	GeoTIFF I/O and raster operations
GIS	GeoPandas/Fiona	Shapefile generation and GIS operations
GIS	Shapely	Polygon geometry and simplification
Metrics	scikit-image	Image analysis utilities
Logging	TensorBoard	Training metric visualization
Data	NumPy/Pandas	Numerical computation and data handling

Table 9: Technology stack

11.1 AI Tools Acknowledgment

We want to be transparent about our use of AI tools during this hackathon. We used **ChatGPT** and **Claude** (Anthropic) to help with several aspects of development:

- Debugging tricky PyTorch errors (especially the weights_only change in PyTorch 2.6)
- Optimizing hyperparameters by discussing strategies for handling class imbalance
- Writing boilerplate code for the Streamlit interface and shapefile export pipeline
- Understanding and implementing the Douglas-Peucker polygon simplification algorithm
- Researching best practices for semantic segmentation on small datasets

While these tools accelerated our development significantly, all architectural decisions, model selection, hyperparameter tuning, data collection, and testing were done by our team. The AI tools served as sophisticated reference tools and pair-programming assistants.

12. Future Scope

While our current system demonstrates the feasibility of automated feature extraction from drone imagery, there are numerous avenues for improvement and expansion:

12.1 Data & Training Improvements

• **Manual Annotation:** Creating a properly annotated dataset of 200-500 images with precise pixel-level labels using tools like CVAT or LabelMe would dramatically improve model accuracy. Our auto-generated labels are a good bootstrap but introduce systematic errors.

• **GPU Training at Higher Resolution:** Training at 512x512 or even 1024x1024 on a GPU would allow the model to capture finer spatial details, improving detection of small buildings and narrow roads. We estimate this could improve IoU by 10-15%.

• **More Training Data Diversity:** Collecting images from different regions, seasons, lighting conditions, and drone altitudes would improve model generalization.

12.2 Architecture Improvements

• **Instance Segmentation:** Moving from semantic to instance segmentation (e.g., Mask R-CNN) would allow counting individual buildings rather than just identifying building regions. This is crucial for property card generation where each building needs a unique identifier.

• **Panoptic Segmentation:** Combining semantic and instance segmentation for a complete scene understanding approach.

• **Attention Mechanisms:** Adding self-attention or transformer-based modules (e.g., SegFormer) could improve the model's ability to capture long-range spatial relationships.

12.3 Integration & Deployment

• **GIS System Integration:** Full integration with Survey of India's GIS infrastructure, including support for proper Coordinate Reference Systems (CRS), GeoTIFF output, and direct QGIS plugin development.

• **Mobile Application:** A lightweight mobile app for field workers to capture drone imagery, run on-device inference, and upload results to a central server.

• **Edge Deployment on Drones:** Deploying a quantized version of the model directly on drone hardware (using NVIDIA Jetson or similar edge devices) for real-time processing during flight.

• **Cloud-Based Processing Pipeline:** A scalable cloud deployment using GPU instances for batch processing of village orthomosaics.

• **Active Learning Pipeline:** A feedback loop where human operators correct model predictions, and these corrections are used to continuously improve the model.

13. Challenges We Faced

Building this system during a hackathon came with its fair share of challenges. Here's an honest account of the obstacles we encountered and how we addressed them:

13.1 Limited Compute Resources

Our biggest constraint was the lack of GPU access. Training a deep neural network on CPU is painfully slow — each epoch at 256x256 resolution took about 25 minutes, and at 512x512 it would have been over an hour. This forced us to use a smaller input resolution (256x256 instead of 512x512), smaller batch sizes (2 instead of 4), and fewer epochs (14 instead of the planned 150). We partially compensated using gradient accumulation to simulate a larger effective batch size, but the resolution constraint remained the primary bottleneck for accuracy.

13.2 Auto-Generated Label Quality

Our HSV-based auto-labeling was a necessary shortcut, but it introduced several issues. Dark shadows were frequently misclassified as RCC roofs. Certain soil colors in agricultural areas were confused with tiled roofs. Road detection was unreliable in areas with tree canopy cover. And the convex hull approximation for buildings, while better than raw pixel classification, still produced imprecise footprints. We spent an entire night tuning HSV thresholds by trial and error, and the resulting labels are functional but far from perfect.

13.3 Class Imbalance

In our drone imagery, approximately 80-85% of pixels are background (vegetation, bare ground, shadows). Buildings might constitute 10-12%, roads 3-5%, and infrastructure less than 1%. This extreme imbalance means the model can achieve 80%+ accuracy by simply predicting everything as background. We addressed this with Focal + Dice loss and per-class weights, but the model still shows a bias toward background predictions for ambiguous pixels.

13.4 PyTorch 2.6 Breaking Changes

We encountered a frustrating bug when loading saved model checkpoints after upgrading to PyTorch 2.6. The new version changed the default value of the `weights_only` parameter in `torch.load()` from `False` to `True`, which broke our checkpoint loading code. The fix was simple (explicitly setting `weights_only=False`), but debugging it took a while because the error message wasn't immediately clear about the cause.

13.5 Memory Constraints

Running inference on large drone images (4000x3000 pixels) with sliding window and TTA on a machine with limited RAM was challenging. Each 512x512 window generates a probability map of shape (10, 512, 512) as float32, and the full-image accumulation buffer requires (10, 3000, 4000) floats ≈ 460 MB. Combined with the model weights (~75 MB) and intermediate tensors, peak memory usage during inference could exceed 2 GB. We had to carefully manage memory by processing windows sequentially rather than in batches.

14. References

[1] Chen, L.-C., Zhu, Y., Papandreou, G., Schroff, F., & Adam, H. (2018). *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation.* Proceedings of the European Conference on Computer Vision (ECCV), pp. 801-818.

[2] Tan, M., & Le, Q. V. (2019). *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.* Proceedings of the 36th International Conference on Machine Learning (ICML), pp. 6105-6114.

[3] Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollar, P. (2017). *Focal Loss for Dense Object Detection.* Proceedings of the IEEE International Conference on Computer Vision (ICCV), pp. 2980-2988.

[4] Ministry of Panchayati Raj, Government of India. *SVAMITVA Scheme: Survey of Villages Abadi and Mapping with Improvised Technology in Village Areas.* https://svamitva.nic.in/

[5] Yakubovskiy, P. (2020). *Segmentation Models Pytorch.* GitHub repository: https://github.com/qubvel/segmentation_models.pytorch

[6] Ronneberger, O., Fischer, P., & Brox, T. (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation.* Medical Image Computing and Computer-Assisted Intervention (MICCAI), pp. 234-241.

[7] Buslaev, A., Iglovikov, V., Khvedchenya, E., Parinov, A., Druzhinin, M., & Kalinin, A. A. (2020). *Albumentations: Fast and Flexible Image Augmentations.* Information, 11(2), 125.

[8] Douglas, D. H., & Peucker, T. K. (1973). *Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature.* Cartographica, 10(2), 112-122.

[9] Loshchilov, I., & Hutter, F. (2016). *SGDR: Stochastic Gradient Descent with Warm Restarts.* arXiv preprint arXiv:1608.03983.

[10] Survey of India. *Large Scale Mapping and Drone Surveys for SVAMITVA.* https://www.surveyofindia.gov.in/

Digital University Kerala (DUK) | AI Model developed by DUK Students

Built with passion, caffeine, and a whole lot of debugging.