Final Report on SKKU Facade Segmentation Net

Deyi Wang

December 2025

(a)

$$\left\{ c^{[2]}_{j,k} \in \mathbb{R}^{81} \;:\; j = 1, \dots, 5,\; k = 1, \dots, 3 \right\}$$

$$x^{[2]}_j \;=\; \sum_{k=1}^{3} \left(c^{[2]}_{j,k} * z^{[1]}_k \right), \qquad j = 1, \dots, 5$$

$$z^{[2]}_j \;=\; \mathrm{ReLU}\left(x^{[2]}_j \right), \qquad j = 1, \dots, 5$$

$$\#\text{params(layer 1)} \;=\; 3 \cdot 9 \cdot 9 \;=\; 3 \cdot 81 \;=\; 243$$

$$\#\text{params(layer 2)} \;=\; 5 \cdot 3 \cdot 9 \cdot 9 \;=\; 15 \cdot 81 \;=\; 1215$$

$$\#\text{params(total)} \;=\; 243 + 1215 \;=\; 1458$$

(b)

For semantic segmentation, the network must output a *class score vector per pixel.* Let the last hidden feature map be $F \in \mathbb{R}^{D \times H \times W}$, and denote the D-dimensional feature vector at pixel (i, j) as $f_{ij} \in \mathbb{R}^D$. A natural last layer is a shared linear classifier applied independently to every spatial location:

$$s_{ij} = W f_{ij} + b, \quad W \in \mathbb{R}^{C \times D},\; b \in \mathbb{R}^C,$$

where $C = 5$ is the number of classes and $s_{ij} \in \mathbb{R}^C$ are the logits. In CNN form, this is exactly a 1×1 convolution with D input channels and C output channels (weight sharing across (i, j) preserves translation equivariance). A softmax over the class dimension converts logits to per-pixel class probabilities.

In my implementation, the last layer is `nn.Conv2d(32, 5, kernel_size=1)` (see `Networks/UNetMini.py`), which matches the linear-algebra view above.

(c)

Architecture I use a U-Net style encoder–decoder with skip connections, and add a lightweight Transformer bottleneck to increase the receptive field and model long-range interactions at low spatial resolution. The model operates on 256×256 inputs and outputs 256×256 logits for 5 classes. Detailed implementation is in `Networks/UNetMini.py`.

Stage	Resolution	Channels / operator
Input	256×256	3
Encoder 1	256×256	DoubleConv $3 \rightarrow 32$, MaxPool 2×2
Encoder 2	128×128	DoubleConv $32 \rightarrow 64$, MaxPool 2×2
Encoder 3	64×64	DoubleConv $64 \rightarrow 128$, MaxPool 2×2
Encoder 4	32×32	DoubleConv $128 \rightarrow 256$, MaxPool 2×2
Bottom	16×16	DoubleConv $256 \rightarrow 512$
Transformer bottleneck	16×16	$2\times$ TransformerEncoderLayer (d=512, 4 heads)
Decoder 1	32×32	UpConv $512 \rightarrow 256$, concat skip, DoubleConv $512 \rightarrow 256$
Decoder 2	64×64	UpConv $256 \rightarrow 128$, concat skip, DoubleConv $256 \rightarrow 128$
Decoder 3	128×128	UpConv $128 \rightarrow 64$, concat skip, DoubleConv $128 \rightarrow 64$
Decoder 4	256×256	UpConv $64 \rightarrow 32$, concat skip, DoubleConv $64 \rightarrow 32$
Output	256×256	1×1 Conv $32 \rightarrow 5$ (logits)

Table 1: Detailed architecture of the submitted model (`Networks/UNetMini.py`). DoubleConv denotes (Conv3×3 + BN + SiLU)×2 + Dropout2d.

Training setup All training hyperparameters are defined in `Train/UNetMini.py`:

- Batch size: 16; epochs: 100.
- Optimizer: AdamW; initial/max learning rate: 5×10^{-4}.
- Scheduler: OneCycleLR (linear anneal), with `pct_start`=0.09.
- Loss: class-weighted cross-entropy to address class imbalance, using weights $1/\sqrt{p_c}$ with p_c estimated from pixel ratios ($[1.679, 1.570, 6.509, 2.462, 4.432]$ for classes 0–4).
- Metric: AP per class via torchmetrics AveragePrecision; selection uses mean AP across classes.

Train/validation loss curves. The code trains on the provided `train` split and uses `test_dev` as a test set. Figure 1 shows the training and validation loss, and the validation mean AP across epochs. For plotting a result file, see `Train/Results/Board.ipynb`.

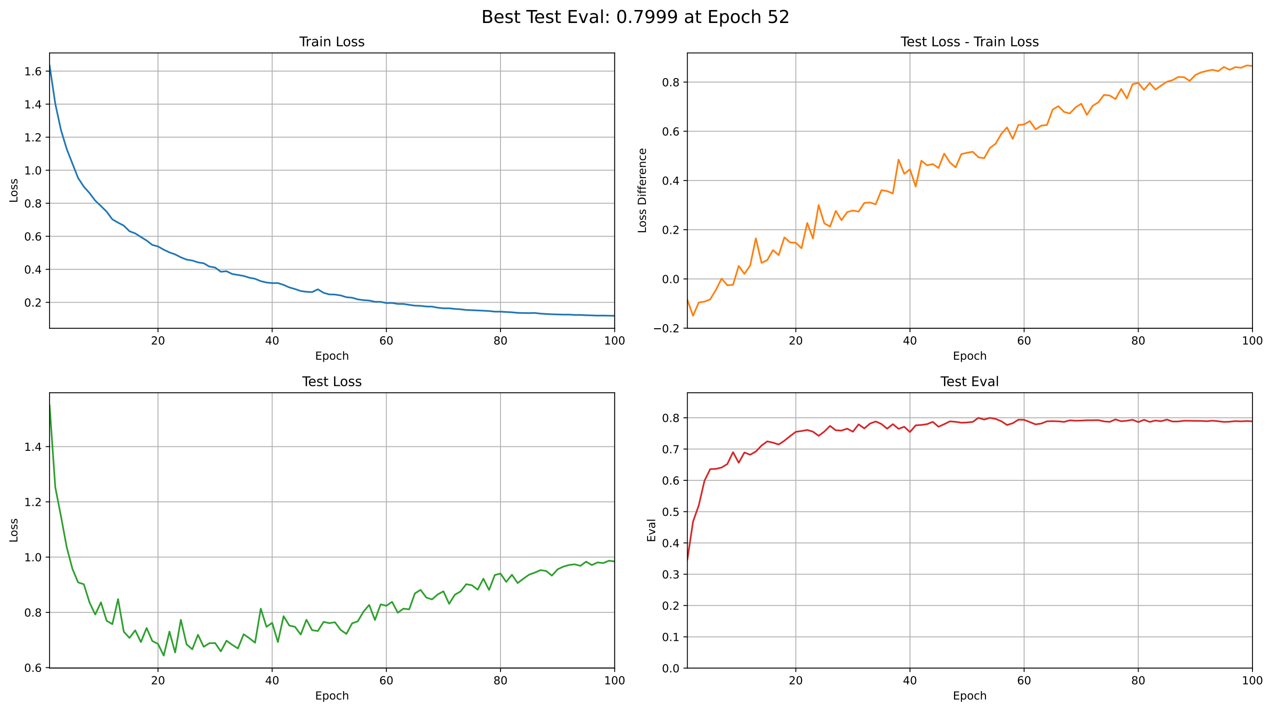

Figure 1: Training curves from `Train/Results/UNetMini/20251214_235701_403_bc1bd0d4.csv`. Left: cross-entropy loss. Right: mean AP (mean over 5 classes).

(d)

The best-performing combination of model definition, optimizer, and training parameters is implemented in:

- Model: `Networks/UNetMini.py` (U-Net + Transformer bottleneck).
- Training entry: `Train/UNetMini.py` (AdamW + OneCycleLR + weighted CE).
- Training loop / evaluation utilities: `Train/lib/train.py`, `Train/lib/evaluation.py`, `Train/lib/criterion.py`.

To run the training and test, just run `Train/UNetMini.py` as the main file after setting up the dataset in `Datasets/MiniFacade.py`.

All the files mentioned above are already included in the submission package on iCampus, but the full codebase is also available at GitHub `https://github.com/realJiaoKan/SKK-Facade-Segmentation-Net` for reference.

(e)

Using `test_dev` as the test split, the best test mean AP is **0.7999** at epoch **52** of run `bc1bd0d4`. The per-class AP at the best epoch is:

Class	AP on test
others (0)	0.8098
facade (1)	0.8644
pillar (2)	0.5521
window (3)	0.9312
balcony (4)	0.8419

Table 2: Test AP (best epoch) computed from the training log CSV.

And the detailed validation, test metrics at the best epoch are could be found in the CSV file. The summary figure has been shown before in Figure 1.

(f) Qualitative result on a custom photo

I use the same preprocessing as the dataset (resize to 256×256, normalize to $[-1, 1]$) and run inference with the saved checkpoint (`Train/Results/UNetMini/Checkpoints/best_latest.pt`). Figure 2 shows an example qualitative result produced by `demo.ipynb`.

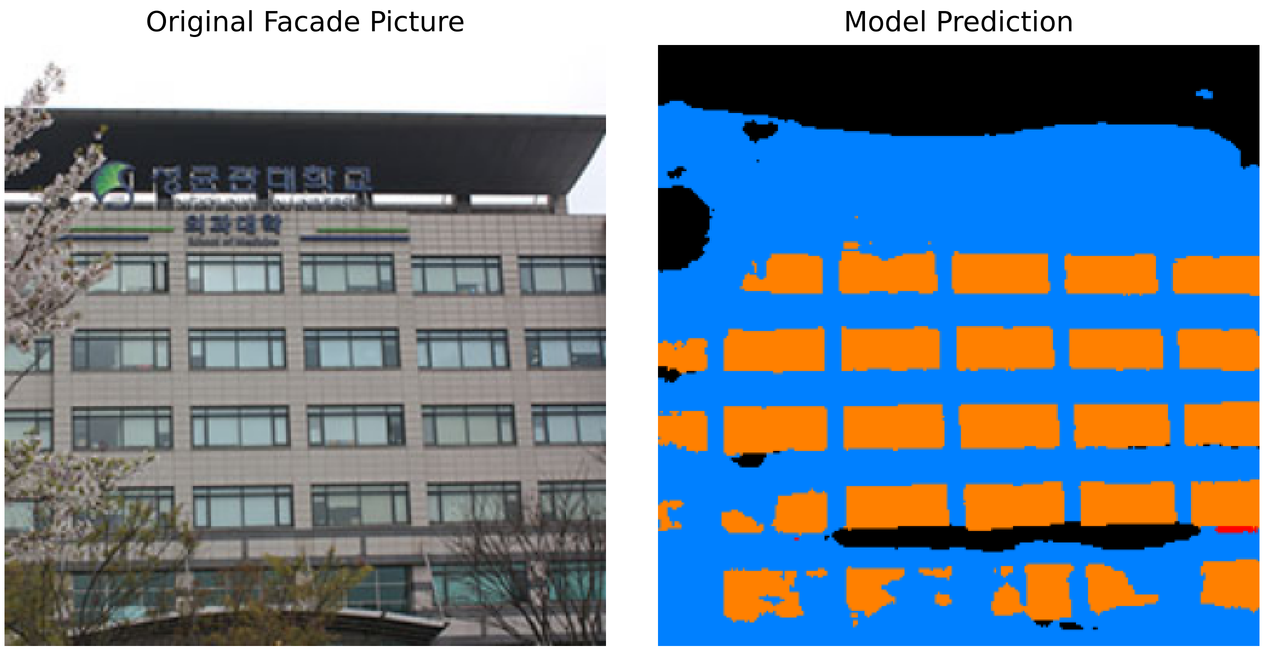

Figure 2: Example (from `demo.ipynb`) showing the model prediction compared to ground truth on a held-out sample. For the final submission requirement, replace the input with a self-taken SKKU building photo and save it as `skku.jpg`, and export the predicted label image using the same palette format.

Commentary Qualitatively, the model produces coherent and visually consistent segmentation results on the SKKU building image. Large facade regions are correctly identified, and repetitive structures such as windows are clearly and regularly segmented. This indicates that the network has learned meaningful architectural patterns and can effectively combine local texture information with global structural cues, aided by the U-Net skip connections and the transformer bottleneck that enlarges the receptive field.

The model works well mainly because the input image resembles the training distribution: a frontal viewpoint, clear facade geometry, and regularly arranged windows. Potential failure cases may occur in regions with occlusions (very clear for the trees in this sample), strong perspective distortion, or architectural styles not seen during training, where the learned geometric and texture features no longer match the visual statistics of the scene.

PS

Repository structure To make my experiments easy to reproduce, the codebase is organized as follows:

- `Datasets/`: dataset preprocessing and PyTorch dataloaders.
- `Networks/`: model definitions (my final model is `Networks/UNetMini.py`; other variants are kept for ablation).

- `Train/`: runnable training entry points and training utilities.
- `Train/lib/`: training loop (`train.py`), loss functions (`criterion.py`), and evaluation metrics (`evaluation.py`).
- `settings.py`: global constants such as `DEVICE`, random seed, and `N_CLASS`.
- `Train/Results/`: saved checkpoints and logs (CSV) for each run, plus a notebook to visualize curves.

Dataset preprocessing and caching The dataset entry is `Datasets/MiniFacade.py`. It reads paired files `ee616_%04d.jpg` and `ee616_%04d.png` under `Datasets/Data/train/` and `Datasets/Data/test_dev/` (or `Datasets/Data/test/` for the real test set), then:

- normalizes images from uint8 $[0, 255]$ to float $[-1, 1]$ and changes layout from (H, W, C) to (C, H, W);
- converts the indexed-color PNG labels into a one-hot tensor of shape $(5, H, W)$;
- caches the processed arrays as `Datasets/Data/train.npz`, `Datasets/Data/test_dev.npz` to speed up future runs (will auto-generate the `.npz` if it is missing).

Using the real `test` split (instead of `test_dev`) By default, the dataloader uses `test_dev` as the "test" split because that is the split provided to us. The switch is controlled by `FLAG_DEV` in `Datasets/MiniFacade.py`:

- Place the real test data under `Datasets/Data/test/`.
- Ensure the file naming follows the same index convention: `ee616_0000.jpg/.png`, `ee616_0001.jpg/.png`, ..., and the indices are contiguous.
- Set `FLAG_DEV=False` so `load_dataset()` will load `test` rather than `test_dev`.
- Set `N_SIZE["test"]` to the *number of samples* in the test folder (equivalently, the maximum index + 1). This is required because `load_raw()` iterates `range(N_SIZE[split])`.

Training, evaluation, and result logging My final training entry is `Train/UNetMini.py`, which wires together:

- model: `Networks/UNetMini.Network`
- data: `Datasets/MiniFacade.load_loader` (train + test/`test_dev`)
- loss: `Train/lib/criterion.get_criterion` (default: class-weighted cross-entropy)
- metric: per-class Average Precision computed in `Train/lib/evaluation.py` (selection uses mean AP over the 5 classes)
- training loop: `Train/lib/train.run` with AdamW and OneCycleLR

Each run saves (1) a best checkpoint to `Train/Results/UNetMini/Checkpoints/` and (2) a CSV log to `Train/Results/UNetMini/` containing train/test loss and per-class AP per epoch (the plotting notebook is `Train/Results/Board.ipynb`).

Minimal reproduction steps

1. Set up a Python 3.9 (3.9.25 in my setup) virtual environment.
2. Make sure using the root path of the repository as the Python working directory (except for notebooks), I recommend to us `"PYTHONPATH": "$workspaceFolder"` to set it using vscode. You can check `.vscode` folder for an example, which is included in the repository.
3. Install dependencies: `pip install -r requirements.txt`.

4. Put the dataset under `Datasets/Data/train/` and `Datasets/Data/test_dev/` (or `test/` if available) following the naming convention described above.
5. Run training: `python Train/UNetMini.py`. Results are written under `Train/Results/UNetMini/`.
6. Visualize curves with `Train/Results/Board.ipynb`, and run qualitative inference with `demo.ipynb`.