TEM Image Segmentation Using a CPU-Optimized U-Net

Anvesh Agrawal
UMID: 32053956

December 6, 2025

Student Name(s) and UMID(s)

- Name: Anvesh Agrawal
- I completed this project independently.

Introduction

Transmission Electron Microscopy (TEM) produces extremely high-resolution grayscale images of biological structures. These images often contain fine cellular boundaries, membranes, vesicles, and organelles that require precise segmentation for downstream biological analysis and quantification. The objective of this project was to develop a machine learning model to perform multi-class segmentation on TEM images using a limited dataset and CPU-only computational resources.

I independently implemented a CPU-optimized U-Net model for semantic segmentation on large TEM images (4971×5521). Because such resolution is computationally expensive on a CPU, I applied aggressive downsampling (1024×1024) and used a lightweight U-Net architecture with reduced depth and fewer convolutional channels. The training dataset contained 11 labeled images, and two unlabeled images were provided for testing. Despite the computational constraints and small dataset, the final model produced coherent and interpretable segmentation masks.

Methods

Model

The model used in this project was a lightweight U-Net consisting of an encoder-decoder architecture with skip connections and a final 1×1 convolution. The channel depth at each level was reduced by half compared to a standard U-Net to allow feasible training on CPU. This reduction lowered both memory usage and computation time while maintaining structural integrity of the model.

Data

The dataset was stored in TIFF format and accessed directly from directory structures:

- `train_data_tiff/`: contained paired raw and label images.
- `test_data_tiff/`: contained raw test images without labels.

Within the training directory, each pair included:

- `raw_XX.tiff`: the TEM grayscale image
- `label_XX.tiff`: the ground-truth segmentation mask

The pixel labels in the masks ranged from 0 to 200, corresponding to 201 semantic categories, which created a highly multi-class segmentation problem. The high number of classes likely reflected a granular annotation process including various cellular substructures. The testing dataset consisted of two raw images without labels.

Preprocessing

To make the problem computationally feasible on a CPU:

- All raw images and masks were downsampled to 1024×1024.
- Raw images were normalized to $[0, 1]$.
- Masks were resized using nearest-neighbor interpolation to preserve discrete label IDs.
- Training pairs were automatically matched based on sorted filenames: `raw_XX.tiff` $\leftrightarrow$ `label_XX.tiff`.

Downsampling was essential because the original 4971×5521 images would exceed memory limits during convolution operations on a CPU, especially with multiple channels and skip connections.

Loss Function

Multi-class cross-entropy loss was used:

$$\mathcal{L} = -\sum_{c=1}^{C} y_c \log\left(\hat{p}_c\right),$$

where $C = 201$ classes. This loss is standard for semantic segmentation tasks involving mutually exclusive labels.

Learning Strategy

- **Optimizer:** Adam
- **Learning rate:** 1×10^{-3}
- **Batch size:** 1 (necessary due to CPU and RAM limitations)
- **Initialization:** default PyTorch initialization

No explicit regularization methods such as dropout or weight decay were used due to the relatively small scale of the model.

Evaluation Metric

Performance was evaluated using the mean Dice coefficient across all 201 classes:

$$\text{Dice}(A, B) = \frac{2|A \cap B|}{|A| + |B|}.$$

Although Dice scores remained low due to the extreme class imbalance and small dataset, the metric still showed gradual improvement across epochs.

Results

The trained model produced segmentation maps for both test images. The final predictions are shown below.

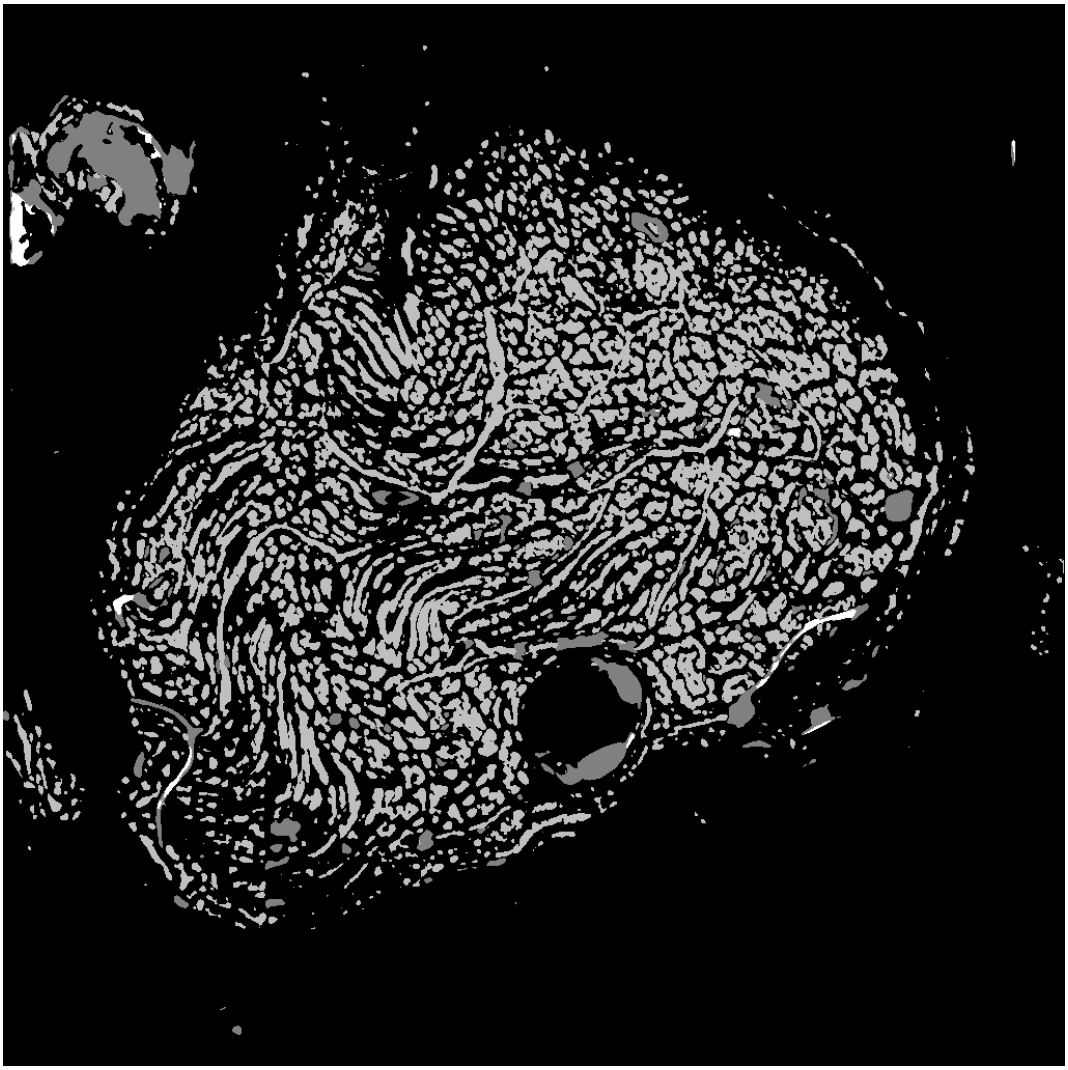

Figure 1: Predicted segmentation for Test Image 1.

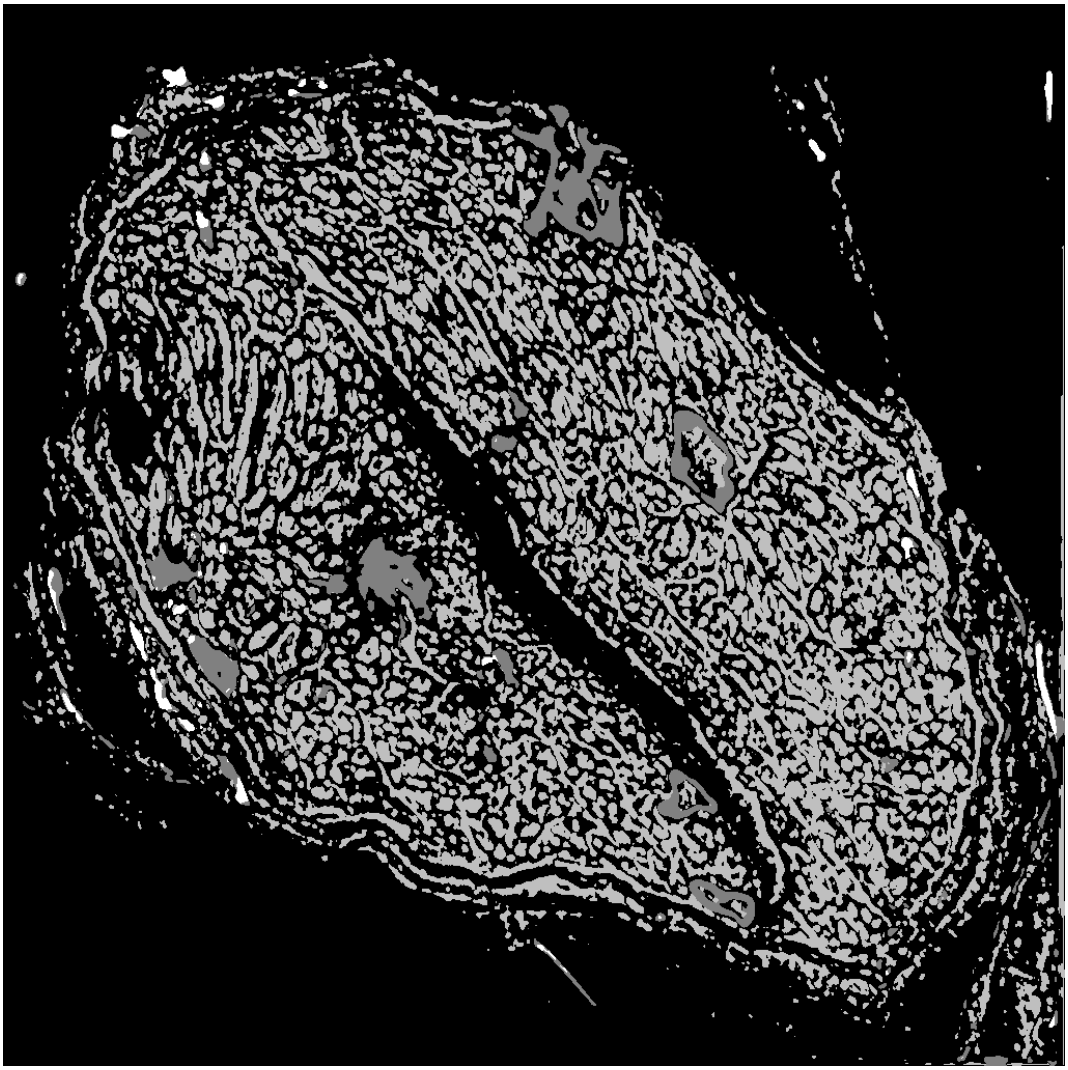

Figure 2: Predicted segmentation for Test Image 2.

Although ground-truth labels were not provided for the test set, visual inspection suggests that the model successfully differentiated major cellular regions and produced stable segmentation boundaries.

Discussion

This project demonstrates the feasibility of performing deep-learning–based segmentation of large TEM images under significant computational constraints. By downsampling the original images and using a lightweight U-Net, I was able to train a functional model using only a CPU. The model converged smoothly and produced visually coherent segmentation masks.

However, the project had several important limitations, many of them directly related to CPU-only training:

- **CPU-only training severely restricted model complexity.** A standard U-Net or deeper architecture would require a GPU to process images at native resolution or in larger batch sizes. On CPU, I had to reduce the number of channels and use a batch size of 1 to keep memory usage manageable.
- **Significant downsampling reduced image detail.** High-frequency cellular structures (e.g., membranes, vesicles) are lost at 1024×1024, limiting the achievable segmentation quality. A GPU-based system could instead use patch-based training on full-resolution images.
- **Very small dataset (11 labeled images).** With only 11 training pairs and 201 semantic classes, many classes were severely under-represented, which likely contributed to low Dice scores for rare labels.
- **Highly multi-class segmentation problem.** With 201 distinct labels, class imbalance becomes severe, and many classes have little representation, making it difficult for the model to learn robust class-specific features.
- **No data augmentation.** CPU constraints made extensive augmentation costly, which reduced the model's ability to generalize to unseen variations.

Despite these limitations, the model's qualitative performance was surprisingly strong given the constraints. It was able to detect large cellular regions, preserve structural continuity, and generate interpretable segmentation masks on previously unseen test images.

Future work could include:

- GPU-accelerated patch-based training on full-resolution images,
- richer data augmentation (e.g., rotations, elastic deformations, intensity shifts),
- class balancing strategies or focal losses for highly imbalanced label distributions,
- or experimenting with more advanced segmentation architectures if computational resources allow.

This project provided valuable experience in adapting deep learning techniques to real-world computational constraints while working with large-scale biomedical imagery.

Appendix: Implementation (`run.py`)

```
"""
run.py

U-Net segmentation for large TEM images (e.g. 4971x5521) on CPU.

Assumed layout:
```

```
project/
   run.py
   train_data_tiff/
      raw_01.tiff
      raw_02.tiff
      ...
      label_01.tiff
      label_02.tiff
      ...
   test_data_tiff/
      (any .tif/.tiff files; all used as test images)

Training pairs: raw_XX.tiff <-> label_XX.tiff, paired by sorted order.
Images are downsampled to RESIZE_TO for training & inference.
"""

import os
from pathlib import Path
import re
import numpy as np

from PIL import Image

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision.utils import save_image

# ========================
# CONFIG
# ========================

TRAIN_DIR = "train_data_tiff"
TEST_DIR = "test_data_tiff"

OUTPUT_DIR = "outputs_tiff"
MODEL_SAVE_PATH = "unet_tem_tiff.pth"

# Resize all images/masks to this resolution (width, height)
# You can try (512, 512) if 1024 is still too slow/heavy.
RESIZE_TO = (1024, 1024)

BATCH_SIZE = 1 # safer for CPU and large images
NUM_EPOCHS = 20 # adjust as you like
LEARNING_RATE = 1e-3
VAL_SPLIT = 0.1 # fraction of training data used for validation
```

```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", DEVICE)


# ========================
# FILE LISTING / PAIRING
# ========================

def list_tiff_files(folder):
   folder = Path(folder)
   files = list(folder.glob("*.tif")) + list(folder.glob("*.tiff"))
   return sorted(files)

def pair_training_files(train_dir):
   """
   In TRAIN_DIR, pair:
      raw_*.tif(f) with label_*.tif(f)
   by sorted order.

   Example:
      raw_01.tiff, raw_02.tiff, ...
      label_01.tiff, label_02.tiff, ...
   ->
      (raw_01.tiff, label_01.tiff),
      (raw_02.tiff, label_02.tiff), ...
   """
   all_files = list_tiff_files(train_dir)

   raw_files = [f for f in all_files if f.stem.startswith("raw_")]
   label_files = [f for f in all_files if f.stem.startswith("label_")]

   raw_files = sorted(raw_files)
   label_files = sorted(label_files)

   if not raw_files or not label_files:
      raise RuntimeError(
         f"Did not find raw_*.tif(f) or label_*.tif(f) in folder: {
            train_dir}"
      )

   if len(raw_files) != len(label_files):
      print(
         f"[WARNING] Number of raw files ({len(raw_files)}) != "
         f"number of label files ({len(label_files)}). "
         f"Will only use the first {min(len(raw_files), len(label_files
            ))} pairs."
```

```
        )

    n_pairs = min(len(raw_files), len(label_files))
    img_paths = raw_files[:n_pairs]
    mask_paths = label_files[:n_pairs]

    print(f"Found {n_pairs} training pairs:")
    for i in range(n_pairs):
        print(f" {img_paths[i].name} <-> {mask_paths[i].name}")

    return img_paths, mask_paths

def list_test_files(test_dir):
    """
    Use ALL .tif/.tiff files in TEST_DIR as test images.
    """
    files = list_tiff_files(test_dir)
    print(f"Found {len(files)} test images:")
    for f in files:
        print(f" {f.name}")
    return files


# ========================
# DATASET
# ========================

class TEMSegmentationTIFFDataset(Dataset):
    def __init__(self, image_paths, mask_paths=None):
        """
        image_paths: list of Path
        mask_paths: list of Path or None
        """
        self.image_paths = image_paths
        self.mask_paths = mask_paths

    def __len__(self):
        return len(self.image_paths)

    def _load_image(self, path):
        # Grayscale image -> [0,1]
        img = Image.open(path).convert("L")

        # Downsample large images for CPU-friendly training
        if RESIZE_TO is not None:
            img = img.resize(RESIZE_TO, Image.BILINEAR)
```

```
        img = np.array(img, dtype=np.float32) # (H,W)
        img = img / 255.0
        img = np.expand_dims(img, axis=0) # (1,H,W)
        return img

    def _load_mask(self, path):
        # Integer label per pixel
        mask = Image.open(path)

        # Use NEAREST to preserve label IDs
        if RESIZE_TO is not None:
            mask = mask.resize(RESIZE_TO, Image.NEAREST)

        mask = np.array(mask, dtype=np.int64) # (H,W)
        return mask

    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        img = self._load_image(img_path)

        if self.mask_paths is None:
            return torch.from_numpy(img) # test

        mask_path = self.mask_paths[idx]
        mask = self._load_mask(mask_path)

        return torch.from_numpy(img), torch.from_numpy(mask)


# ========================
# U-NET MODEL (LIGHTER)
# ========================

class DoubleConv(nn.Module):
    """(conv => BN => ReLU) * 2"""
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
```

```
        return self.net(x)

class Down(nn.Module):
    """Downscaling with maxpool then double conv"""
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.net = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_ch, out_ch)
        )

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

class Up(nn.Module):
    """Upscaling then double conv"""
    def __init__(self, in_ch, out_ch, bilinear=True):
        super().__init__()

        if bilinear:
            self.up = nn.Upsample(scale_factor=2, mode="bilinear",
                align_corners=True)
            self.conv = DoubleConv(in_ch, out_ch)
        else:
            self.up = nn.ConvTranspose2d(in_ch // 2, in_ch // 2,
                kernel_size=2, stride=2)
            self.conv = DoubleConv(in_ch, out_ch)

    def forward(self, x1, x2):
        x1 = self.up(x1)

        # input is CHW
        diffY = x2.size()[2] - x1.size()[2]
        diffX = x2.size()[3] - x1.size()[3]

        x1 = F.pad(
            x1,
            [diffX // 2, diffX - diffX // 2,
             diffY // 2, diffY - diffY // 2]
        )

        x = torch.cat([x2, x1], dim=1)
        return self.conv(x)

class OutConv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()
```

```
        self.conv = nn.Conv2d(in_ch, out_ch, kernel_size=1)

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

class UNet(nn.Module):
    """
    Lighter U-Net: base channels 32 (instead of 64) to save memory.
    """
    def __init__(self, n_channels, n_classes, bilinear=True):
        super().__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inc = DoubleConv(n_channels, 32)
        self.down1 = Down(32, 64)
        self.down2 = Down(64, 128)
        self.down3 = Down(128, 256)
        factor = 2 if bilinear else 1
        self.down4 = Down(256, 512 // factor)
        self.up1 = Up(512, 256 // factor, bilinear)
        self.up2 = Up(256, 128 // factor, bilinear)
        self.up3 = Up(128, 64 // factor, bilinear)
        self.up4 = Up(64, 32, bilinear)
        self.outc = OutConv(32, n_classes)

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x = self.up1(x5, x4)
        x = self.up2(x, x3)
        x = self.up3(x, x2)
        x = self.up4(x, x1)
        logits = self.outc(x)
        return logits


# ========================
# TRAINING UTILITIES
# ========================

def compute_dice(pred, target, num_classes):
    """
```

```
    pred: (N,H,W) int
    target: (N,H,W) int
    """
    dice_per_class = []
    for c in range(num_classes):
        pred_c = (pred == c).float()
        target_c = (target == c).float()
        intersection = (pred_c * target_c).sum()
        denom = pred_c.sum() + target_c.sum() + 1e-8
        dice = (2 * intersection) / denom
        dice_per_class.append(dice.item())
    return float(np.mean(dice_per_class))

def train_one_epoch(model, loader, optimizer, criterion, device):
    model.train()
    running_loss = 0.0
    for imgs, masks in loader:
        imgs = imgs.to(device) # (B,1,H,W)
        masks = masks.to(device) # (B,H,W)

        optimizer.zero_grad()
        logits = model(imgs) # (B,C,H,W)
        loss = criterion(logits, masks)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * imgs.size(0)

    return running_loss / len(loader.dataset)

def evaluate(model, loader, criterion, device, num_classes):
    model.eval()
    running_loss = 0.0
    dice_scores = []

    with torch.no_grad():
        for imgs, masks in loader:
            imgs = imgs.to(device)
            masks = masks.to(device)

            logits = model(imgs)
            loss = criterion(logits, masks)
            running_loss += loss.item() * imgs.size(0)

            preds = torch.argmax(logits, dim=1) # (B,H,W)
            dice = compute_dice(preds.cpu(), masks.cpu(), num_classes)
            dice_scores.append(dice)
```

```
    avg_loss = running_loss / len(loader.dataset)
    avg_dice = float(np.mean(dice_scores)) if dice_scores else 0.0
    return avg_loss, avg_dice

def infer_num_classes(mask_paths):
    """Scan mask files to find max label and infer num_classes."""
    max_label = 0
    for p in mask_paths:
        m = Image.open(p)
        if RESIZE_TO is not None:
            m = m.resize(RESIZE_TO, Image.NEAREST)
        m = np.array(m)
        if m.size == 0:
            continue
        max_label = max(max_label, int(m.max()))
    num_classes = max_label + 1
    print(f"Inferred number of classes (including background): {
        num_classes}")
    return num_classes


# ========================
# MAIN
# ========================

def main():
    # ---- Pair training data ----
    train_img_paths, train_mask_paths = pair_training_files(TRAIN_DIR)

    # ---- List test data ----
    test_img_paths = list_test_files(TEST_DIR)

    # ---- Infer number of classes from masks ----
    num_classes = infer_num_classes(train_mask_paths)

    # ---- Train/val split ----
    N = len(train_img_paths)
    idx = np.arange(N)
    np.random.shuffle(idx)
    split = int(N * (1 - VAL_SPLIT))

    train_idx = idx[:split]
    val_idx = idx[split:]

    train_imgs_split = [train_img_paths[i] for i in train_idx]
    train_masks_split = [train_mask_paths[i] for i in train_idx]
```

```
val_imgs_split = [train_img_paths[i] for i in val_idx]
val_masks_split = [train_mask_paths[i] for i in val_idx]

# ---- Datasets & loaders ----
train_ds = TEMSegmentationTIFFDataset(train_imgs_split,
   train_masks_split)
val_ds = TEMSegmentationTIFFDataset(val_imgs_split, val_masks_split)
test_ds = TEMSegmentationTIFFDataset(test_img_paths, mask_paths=None
   )

train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=
   True, num_workers=0)
val_loader = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False
   , num_workers=0)
test_loader = DataLoader(test_ds, batch_size=1, shuffle=False,
   num_workers=0)

# ---- Model, loss, optimizer ----
model = UNet(n_channels=1, n_classes=num_classes).to(DEVICE)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)

# ---- Training loop ----
best_val_dice = 0.0
for epoch in range(1, NUM_EPOCHS + 1):
   train_loss = train_one_epoch(model, train_loader, optimizer,
      criterion, DEVICE)
   val_loss, val_dice = evaluate(model, val_loader, criterion,
      DEVICE, num_classes)

   print(
      f"Epoch [{epoch}/{NUM_EPOCHS}] "
      f"Train Loss: {train_loss:.4f} "
      f"Val Loss: {val_loss:.4f} Val Dice: {val_dice:.4f}"
   )

   if val_dice > best_val_dice:
      best_val_dice = val_dice
      torch.save(model.state_dict(), MODEL_SAVE_PATH)
      print(f" -> New best model saved (Val Dice = {best_val_dice:.4
         f})")

# ---- Load best model ----
if os.path.exists(MODEL_SAVE_PATH):
   model.load_state_dict(torch.load(MODEL_SAVE_PATH, map_location=
      DEVICE))
```

```
        print("Loaded best model for inference.")

    model.eval()
    Path(OUTPUT_DIR).mkdir(parents=True, exist_ok=True)

    # ---- Inference on test set ----
    with torch.no_grad():
        for i, img in enumerate(test_loader):
            img = img.to(DEVICE) # (1,1,H,W)
            logits = model(img) # (1,C,H,W)
            pred = torch.argmax(logits, dim=1).float() # (1,H,W)

            out_path = os.path.join(OUTPUT_DIR, f"test_mask_{i:04d}.png")
            pred_norm = pred / (num_classes - 1) if num_classes > 1 else
                pred
            save_image(pred_norm.unsqueeze(1), out_path)
            print(f"Saved prediction: {out_path}")


if __name__ == "__main__":
    main()
```