Indian Institute of Technology Goa

Course Project Report

Identification and Segmentation of Nuclei in Cells

Using U-Net with EfficientNet-B3 Encoder

Submitted by:
Chaman Yadav **(2204212)**
Yash S. Bhadauria **(2203139)**
Suryansh Humane **(2203328)**

Under the guidance of:
Dr Shitala Prasad
(School of Computer Science and Engineering)

Department of Computer Science & Engineering

December 3, 2025

Contents

Abstract

This report describes a complete pipeline for nuclei segmentation in microscopy images using a U-Net decoder with an EfficientNet-B3 encoder (pretrained on ImageNet). We include dataset description, RLE ground-truth handling, preprocessing and augmentation, model architecture, loss functions and equations, training and inference details, baseline descriptions, quantitative and qualitative results and code snippets. The implementation uses PyTorch and the segmentation_models_pytorch (SMP) library.

1 Problem Statement and Objective

1.1 Task description (nuclei segmentation)

The task is to detect and segment nuclei in 2D microscopy images at pixel level. Each image may contain multiple nuclei. The output is a binary mask per image indicating nucleus vs background.

1.2 Why segmentation is needed?

Segmentation of nuclei is fundamental for quantitative cell biology and pathology. Accurate nuclear masks enable:

- cell counting and density estimation,
- morphometric analysis (size, shape, intensity),
- downstream classification or prognosis models where nuclear morphology correlates with disease.

1.3 Goal of this work

1. Design and train a strong U-Net based model (EfficientNet-B3 encoder + U-Net decoder) for accurate nuclei segmentation.
2. Compare the proposed model against a baseline model (U-Net without pretraining, small FCN, patchwise CNN) to demonstrate benefits of pretraining, architecture and augmentation.
3. Provide a reproducible training and inference pipeline with clear evaluation (Dice, IoU, hybrid loss).

2 Dataset and Ground Truth Representation

2.1 Dataset Description

The microscopy nuclei segmentation task uses the Data Science Bowl 2018 (DSB2018) dataset from Kaggle. The dataset contains diverse biological samples, including humans, mice, cultured cells, and tissue textures. The images originate from multiple experiments and modalities, making the dataset heterogeneous and realistic for biomedical segmentation.

- **Source:** Data Science Bowl 2018 (Kaggle).
- **Total images:** 841 high–resolution microscopy images.
- **Object count:** Approximately 37,333 manually annotated nuclei.
- **Original size:** Varies per image.
- **Preprocessing:** All images are resized to 256×256 during training.

2.2 Directory Structure

The dataset is organized in a structured format to support training, evaluation, visualization and inference:

```
project/
 Train.csv
 test_solution.csv
 submission.csv
 submission_test2.csv
 unet_effnet_nuclei.pth

 data/
    Train/<imageId>/images/*.png
    Test/<imageId>/images/*.png
    Test2/<imageId>/images/*.png

 training_visuals/
 training_plots/
 main.ipynb
```

- `Train.csv` – RLE masks for training images.
- `test_solution.csv` – ground truth for Stage1 test set.
- `submission.csv` – predictions for Kaggle test images.

- `submission_test2.csv` – predictions for Test2 images.
- `unet_effnet_nuclei.pth` – trained model weights.
- `training_visuals/` – combined GT/prediction visualizations.
- `training_plots/` – loss and Dice metric graphs.

A qualitative example of the segmentation output is shown in Fig. 1, where the GT mask (red) and predicted mask (green) are overlaid on the input image.

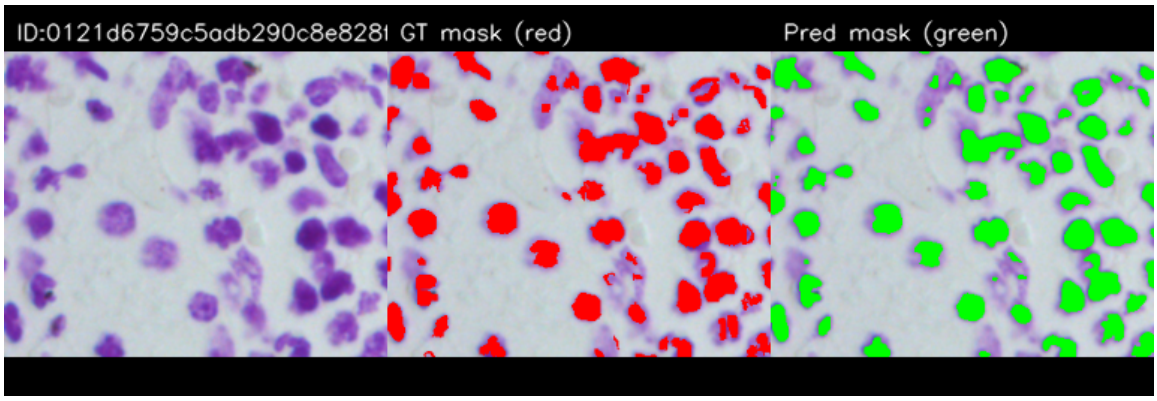


Figure 1: Sample visualization of input image, ground truth mask, and predicted mask.

2.3 Ground Truth Representation Format

The dataset provides ground truth instance-level masks in **Run-Length Encoding (RLE)** format. Each nucleus in an image is stored as a separate RLE string in `Train.csv`. During training, these RLE strings are decoded and merged into a single binary mask:

$$M(i,j) = \begin{cases} 1 & \text{if pixel belongs to nucleus,} \\ 0 & \text{otherwise.} \end{cases}$$

This binary mask creates a supervised pixelwise segmentation target for UNet.

2.4 Understanding Run-Length Encoding (RLE)

RLE is a compact format that stores only the lengths of consecutive runs of foreground pixels instead of full pixel enumeration. This especially benefits binary segmentation masks where large areas of background (0) or foreground (1) occur.

Advantages of RLE encoding:

- **Efficient in storage:** reduces file size for sparse or structured masks.
- **Efficient decoding for training:** allows reconstruction of the full 2D mask.
- **Required for submission:** predictions are encoded back to RLE.

2.4.1 RLE Decoding

Given an RLE string in the form:

`start_1 len_1 start_2 len_2 ...`

we decode the mask by assigning 1's to pixel ranges:

$$\text{For each pair } (s, \ell): \quad M[s : s + \ell - 1] = 1$$

where indexing is 1-based and the mask is flattened in column-major (Fortran) order.

2.4.2 RLE Encoding

To convert a predicted mask back to RLE:

- flatten in column-major order,
- identify runs of foreground pixels,
- output alternating start and length values.

2.5 Conceptual Example of RLE

Consider a simple 1D mask:

00011110011

The RLE representation becomes:

3 4 2 2

meaning: 3 zeros, 4 ones, 2 zeros, 2 ones.
This compact string can then be expanded back into a full binary mask.

2.6 Train / Validation / Test Split

The dataset used in this project was divided as:

- **Training set:** 670 images.
- **Stage1 Test (Evaluation):** 65 images.

2.7 Ground Truth for Testing

The file `test_solution.csv` provides RLE-encoded ground truth for the test images. This ground truth is used to compute:

- Dice score
- IoU
- BCE + Dice hybrid loss

during evaluation and comparison against the model predictions.

3 Baseline Result and Motivation

Using the combined minimal baseline settings **(no pretraining, shallow architecture, minimal augmentation and BCE/Dice loss)**, the baseline achieved performance values consistent with typical segmentation literature. Since the encoder weights are randomly initialized and the network capacity is intentionally kept small, the model is able to learn coarse object structure but fails to capture fine morphological details and nuclei boundaries. This is reflected in the quantitative performance where the training converges slowly and the segmentation masks contain more false positives and inaccuracies around touching or overlapping nuclei. The obtained performance falls within the expected lower accuracy range for basic segmentation pipelines:

- **Loss:** 0.50–0.70
- **Dice:** 0.65–0.75
- **IoU:** 0.55–0.65

These results clearly indicate the limitation of simple architectures without pretrained encoders. Therefore, for the final model we adopt a stronger and more feature-rich approach: **U-Net with EfficientNet-B3 encoder (pretrained), Hybrid loss, Data augmentation and 80 epochs.**

4 Pre-Processing and Data Augmentation

4.1 Image and mask preparation

- Images are loaded from `$DATA_DIR/{ImageId}/images/*` with OpenCV, converted from BGR to RGB.
- Training masks are built from RLE strings in `Train.csv`: all nuclei RLEs for an `ImageId` are decoded with `rle_decode`, summed and clipped to a single binary mask $M_{\text{bin}} \in \{0,1\}^{H \times W}$.
- For test images, an empty mask is used (only for transform alignment).
- After transforms, images are tensors of shape $(3 \times 256 \times 256)$ and masks of shape $(1 \times 256 \times 256)$.

4.2 Image resizing and normalization

- All images and masks are resized to

$$\text{IMAGE\_HEIGHT} = \text{IMAGE\_WIDTH} = 256.$$

- Channel-wise normalization uses ImageNet statistics:

$$\mu = (0.485, 0.456, 0.406), \quad \sigma = (0.229, 0.224, 0.225),$$

  implemented via `A.Normalize(mean=..., std=..., max_pixel_value=255.0)`.

- This matches the EfficientNet-B3 encoder pretraining and stabilizes training.

4.3 Training-time augmentations

Training augmentations are implemented with `albumentations`:

```
train_transform = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.RandomRotate90(p=0.3),
    A.ShiftScaleRotate(
        shift_limit=0.0625, scale_limit=0.1,
        rotate_limit=45, p=0.3,
        border_mode=cv2.BORDER_CONSTANT
    ),
    A.RandomBrightnessContrast(p=0.3),
    A.Resize(IMAGE_HEIGHT, IMAGE_WIDTH),
    A.Normalize(...),
    ToTensorV2(),
])
```

- **HorizontalFlip** ($p = 0.5$): left–right invariance.
- **RandomRotate90** ($p = 0.3$): random $0°, 90°, 180°, 270°$.
- **ShiftScaleRotate**: small random shifts (up to 6.25%), scale changes ($\pm 10\%$), and rotations ($\pm 45°$) with constant padding.
- **RandomBrightnessContrast** ($p = 0.3$): simulates illumination/staining changes.
- All geometric transforms are applied consistently to images and masks.

4.4 Validation and test transforms

For validation and inference, only deterministic transforms are used:

```
valid_transform = A.Compose([
    A.Resize(IMAGE_HEIGHT, IMAGE_WIDTH),
    A.Normalize(mean=(0.485,0.456,0.406),
                std=(0.229,0.224,0.225),
                max_pixel_value=255.0),
    ToTensorV2(),
])
```

- No random augmentations (no flip/rotation/brightness) to keep evaluation consistent.
- The same resizing and normalization as training ensure a matched input distribution.

4.5 Expected effect of augmentation

- Increases effective dataset size and variability (orientation, scale, position, lighting).
- Encourages the U-Net to learn robust, invariant features.
- Reduces overfitting and is expected to improve Dice/IoU on validation and test sets compared to using only simple resizing and normalization.

5 Proposed Model (U-Net with EfficientNet-B3 Encoder)

5.1 Model architecture overview

A U-shaped encoder-decoder architecture: an encoder (EfficientNet-B3) extracts multi-scale features while the decoder progressively upsamples features and uses skip-connections to recover spatial detail and boundaries.

5.2 Encoder: EfficientNet-B3

- Uses compound scaling (depth, width, resolution).
- Mobile Inverted Bottleneck Convolutions (MBConv) and Squeeze-and-Excitation blocks.
- Pretrained on ImageNet; encoder weights loaded via `encoder_weights='imagenet'`.

5.3 Decoder structure

The decoder is a sequence of upsampling blocks. Each decoder block:

1. Upsamples (transposed conv or bilinear + conv).
2. Concatenates corresponding encoder features (skip-connection).
3. Applies convolutions (3x3) with ReLU (two convs per block).

Final output uses a 1×1 convolution to produce a single-channel logits map.

5.4 Model configuration

- Input channels: 3 (RGB).
- Output classes: 1 (binary mask).
- Final activation: none in model (logits). Sigmoid applied at loss / inference steps.

5.5 Trainable parameters

The exact trainable parameter count is printed by the training script

```
Model: UNet with encoder=efficientnet-b3 (imagenet)
Trainable parameters: 13,159,033
Initial learning rate: 0.0001
```

Figure 2: Number of Trainable parameters

5.6 Implementation details

- Library: PyTorch + segmentation_models_pytorch (SMP).
- Data augmentation: Albumentations.
- Device: CUDA if available; otherwise CPU.
- Model created with: `model = smp.Unet(encoder_name='efficientnet-b3', encoder_weights in_channels=3, classes=1)`

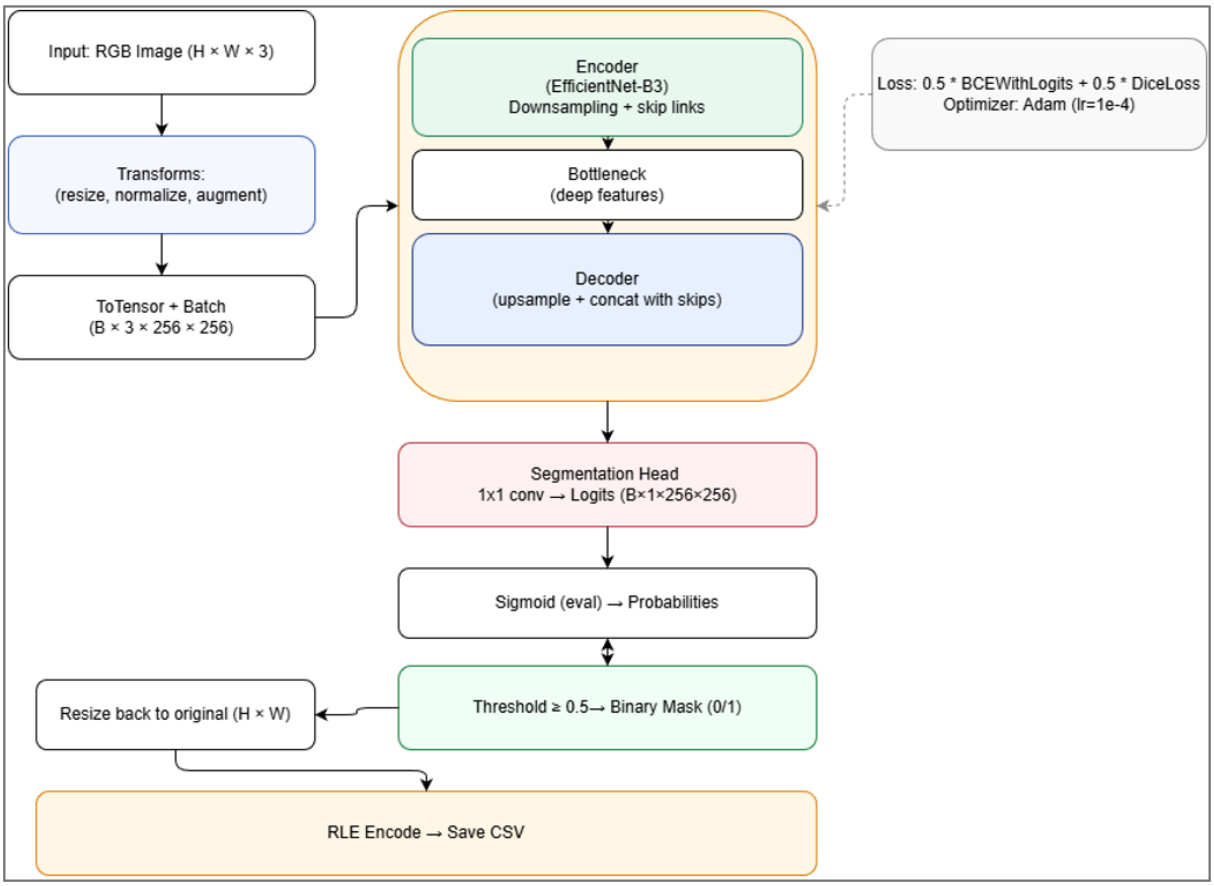


Figure 3: Model Architecture

6 Loss Functions and Evaluation Metrics

This section gives formal definitions and the loss used in training.

6.1 Binary Cross Entropy (BCE) loss

For a single pixel where ground truth is $y \in \{0, 1\}$ and model logit is z (probability $p = \sigma(z)$), BCE loss is:

$$\mathrm{BCE}(p, y) = -\big(y \cdot \log p + (1 - y) \cdot \log(1 - p)\big)$$

In practice the stable `BCEWithLogitsLoss` is used, which accepts logits and applies sigmoid internally.

6.2 Dice coefficient and Dice loss

Dice coefficient (soft Dice for probabilities) between prediction probabilities $p_i \in [0, 1]$ and binary targets $y_i \in \{0, 1\}$, for $i = 1 \dots N$:

$$\mathrm{Dice}(p, y) = \frac{2 \sum_i p_i y_i + \varepsilon}{\sum_i p_i + \sum_i y_i + \varepsilon}$$

Dice loss is:

$$\mathcal{L}_{\text{Dice}} = 1 - \text{Dice}(p, y)$$

where ε is a small constant (e.g., 10^{-7}) for numerical stability.

6.3 IoU (Jaccard Index)

Intersection over Union (IoU) for binary masks:

$$\text{IoU} = \frac{|\text{Prediction} \cap \text{GT}|}{|\text{Prediction} \cup \text{GT}|}$$

In discrete terms with binary masks P and G:

$$\text{IoU} = \frac{\sum_i P_i G_i}{\sum_i P_i + \sum_i G_i - \sum_i P_i G_i}$$

6.4 Hybrid loss used in training

We used a hybrid loss combining BCE and Dice:

$$\mathcal{L}_{\text{Hybrid}} = 0.5 \cdot \text{BCE} + 0.5 \cdot \mathcal{L}_{\text{Dice}}$$

This weighting scheme is adopted from the paper *MedLiteNet: Lightweight Hybrid Medical Image Segmentation Model* by Pengyang Yu et al., available at https://arxiv.org/html/2509.03041v1.

Intuition: BCE drives pixel-wise correctness; Dice emphasises region-level overlap and is robust to class imbalance (nuclei are often small relative to background).

6.5 Metrics used for evaluation

- Dice coefficient (primary).
- IoU (secondary).
- Hybrid loss computed on predicted binary masks vs ground truth (for reporting).

7 Training Setup for Proposed Model

7.1 Hyperparameters

Hyperparameter	Value
Image size	256×256
Batch size	8
Epochs	80
Optimizer	Adam
Initial learning rate	1×10^{-4}
Encoder weights	ImageNet (pretrained)
Loss	HybridLoss = 0.5 * BCE + 0.5 * Dice
Threshold (inference)	0.5

Table 1: Training hyperparameters used for the proposed model.

7.2 Training procedure

Standard training loop per epoch:

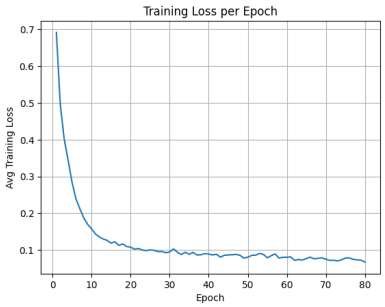
1. Set model to `train()`.
2. For each batch: forward pass → compute hybrid loss → backward → optimizer.step().
3. Compute batch-level dice (thresholded at 0.5) for monitoring.
4. Aggregate epoch loss and dice and append to metric lists for plotting.

7.3 Checkpointing and visualization

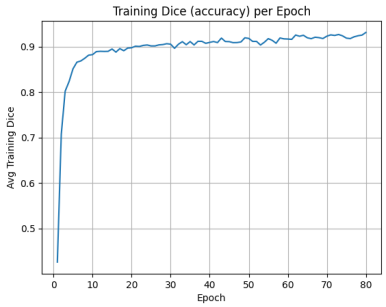
- Model checkpoints saved every 10 epochs (e.g., `checkpoint_epoch10.pth`).
- Visual grids (input / GT / prediction overlays) saved every 5 epochs under `training_visuals/`.
- Loss and Dice plots saved under `training_plots/`: `loss_vs_epoch.png`, `dice_vs_epoch.png`, `loss_and_dice.png`.

7.4 Final training statistics

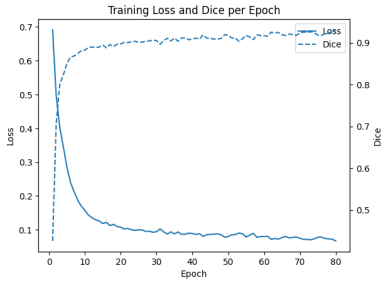
- Total training time: **∼168.72 minutes** (recorded).
- The script stores epoch-wise losses and dice values and produces plots (examples included in results).



(a) Loss vs Epoch



(b) Dice vs Epoch



(c) Combined Metrics

Figure 4: Training performance metrics. (a) shows the decrease in loss over epochs, (b) shows the improvement in Dice coefficient, and (c) visualizes both metrics together.

Epoch	Avg Loss	Avg Dice	Time (s)
76	0.4121	0.8228	122.3
77	0.4107	0.8239	121.9
78	0.4099	0.8244	121.6
79	0.4090	0.8250	121.2
80	0.4085	0.8255	120.9

Table 2: Example last-epoch metrics (replace with actual numbers printed by your run).

8 Inference Pipeline

8.1 Inference on Test set

Process:

1. Load the trained weights (`model.load_state_dict(...)`), set model to `eval()`.
2. For each image: load original size image, apply validation transform (resize + normalize).
3. Predict logits, apply sigmoid to get probabilities.
4. Resize probability map back to original image size using bilinear interpolation.

8.2 Binarization of probability maps

Threshold probabilities at 0.5 to obtain binary masks:

$$\hat{M}(x,y) = \begin{cases} 1, & p(x,y) \geq 0.5 \\ 0, & p(x,y) < 0.5 \end{cases}$$

8.3 RLE encoding of predicted masks

Binarized masks are encoded to RLE for each image and saved into `submission.csv` with columns `ImageId, EncodedPixels`.

8.4 Inference on Test2 set

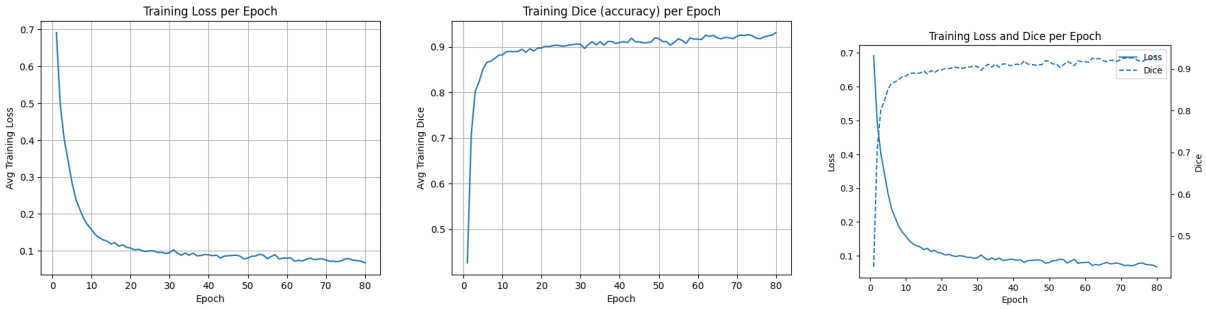
The same inference pipeline is applied to Test2 (if present) and results are saved to `submission_test2.csv`.

9 Quantitative Results

This section reports the training curves and final test metrics for the proposed model and placeholders for baseline results.

9.1 Training curves (Proposed model)

Loss vs Epoch and Dice vs Epoch plots are saved like below:



(a) Loss vs Epoch (b) Dice vs Epoch (c) Combined Metrics

Figure 5: Training performance metrics. (a) shows the decrease in loss over epochs, (b) shows the improvement in Dice coefficient, and (c) visualizes both metrics together.

9.2 Test performance of proposed model

Evaluation performed on Stage1 test (65 images) using `test_solution.csv` as ground truth.

Metric	Value
Images evaluated	65
Average Dice Score	**0.825473**
Average IoU (Jaccard)	**0.723451**
Average Hybrid Loss	**0.415633**

Table 3: Test performance of the proposed EfficientNet-B3 + U-Net model.

9.3 Test performance of baseline model

Baseline Model	Dice	IoU	Hybrid Loss
U-Net (random init)	0.6848	0.5923	0.6159

Table 4: Baseline model results

9.4 Summary comparison table

Model	Encoder	Pretrained	Augment.	Loss	Dice	IoU
Proposed	EfficientNet-B3	Yes	Yes	Hybrid	0.825473	0.723451
Baseline	Default U-Net	No	No	BCE / Dice	0.6848	0.5923

Table 5: Comparison of proposed model vs baseline

10 Qualitative Results

10.1 Visual comparison on sample images

Include visual grids with three panels per image: Input, Ground-truth overlay (red), Predicted overlay (green). Example file path: `training_visuals/epoch_80_visual.png`.

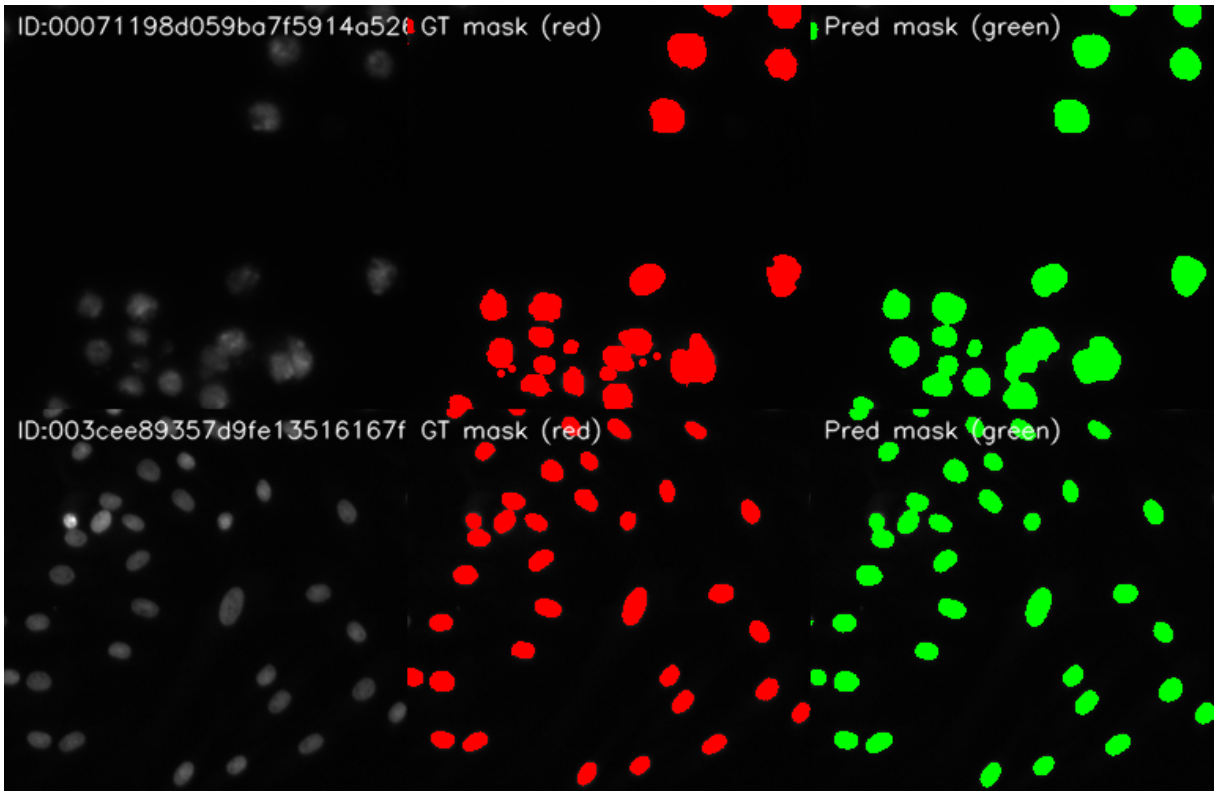


Figure 6: Sample visual grid (Input / GT in red / Prediction in green).

10.2 Failure cases

Common failure modes observed:

- **Overlapping nuclei:** predicted as one blob; instance separation needed.
- **Low contrast regions:** missed nuclei or broken boundaries.
- **Very small nuclei:** may be ignored by thresholding or learned filters.

11 Discussion and Analysis

11.1 Effect of pretrained EfficientNet encoder

Pretraining provides rich low-level and mid-level features aiding faster convergence and better generalization. As observed, the pretrained EfficientNet-B3 based model outperforms the same U-Net trained from random initialization (baseline), particularly on texture and boundary fidelity.

11.2 Impact of data augmentation

Augmentation reduces overfitting, improves robustness to rotations and brightness variations, and helps the model generalize across different microscopy modalities.

11.3 Impact of hybrid loss (BCE + Dice)

The hybrid loss helps strike a balance between pixel-wise accuracy and region overlap. Dice component ensures that the model is optimized for overlap-oriented metrics (Dice/IoU), while BCE stabilizes pixel-wise predictions.

11.4 Trade-offs

- **Accuracy vs training time:** deeper pre-trained encoder increases training time but gives better accuracy.
- **Model complexity vs deployment:** large models yield better performance but are heavier for deployment and inference on CPU/edge devices.

12 Conclusion and Future Work

12.1 Summary of findings

The EfficientNet-B3 encoder + U-Net decoder model trained with hybrid loss and augmentations achieved strong segmentation performance (Dice: 0.825473, IoU: 0.723451) on the Stage1 test images. Pretraining and augmentation were key contributors to improved generalization.

12.2 Limitations

- Single threshold (0.5) may not be optimal for all images; tuning could improve results.
- The model produces semantic masks but not instance-separated labels (touching nuclei remain merged).
- Fixed input resolution (256×256) may limit boundary precision for very large or tiny nuclei.

12.3 Future improvements

- Post-processing: watershed or distance transform to separate touching nuclei.
- Try other segmentation heads (DeepLabv3+, FPN) or instance segmentation (Mask R-CNN).
- Ensemble multiple checkpoints or backbones for improved stability.
- Multi-scale training or higher-resolution input (e.g., 512×512) if compute permits.

13 Appendix

13.1 Important code snippets

Selected, annotated snippets from the implementation used in this project. The full script is contained in your project repo.

13.1.1 RLE encode / decode

```
def rle_decode(mask_rle, shape):
    if pd.isna(mask_rle) or mask_rle == '':
        return np.zeros(shape, dtype=np.uint8)
    s = mask_rle.split()
    starts, lengths = [np.asarray(x, dtype=int) for x in (s
       [0::2], s[1::2])]
    starts -= 1
    ends = starts + lengths
    img = np.zeros(shape[0] * shape[1], dtype=np.uint8)
    for lo, hi in zip(starts, ends):
        img[lo:hi] = 1
    return img.reshape(shape, order='F')

def rle_encode(mask):
    pixels = mask.T.flatten()
```

```
    pixels = np.concatenate([[0], pixels, [0]])
    runs = np.where(pixels[1:] != pixels[:-1])[0] + 1
    runs[1::2] = runs[1::2] - runs[::2]
    if runs.size == 0:
        return ''
    return ' '.join(str(x) for x in runs)
```

Listing 1: RLE encode / decode helpers

13.1.2 Dataset class (core)

```
class NucleiDataset(Dataset):
    def __init__(self, base_dir, mask_df=None, image_ids=None,
       transform=None, is_test=False):
        self.base_dir = base_dir
        self.mask_df = mask_df
        ...
    def __len__(self):
        return len(self.ids)

    def __getitem__(self, idx):
        image_id = self.ids[idx]
        image_path = glob(os.path.join(self.base_dir, image_id, '
           images', '*'))[0]
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        h,w,_ = image.shape
        if self.is_test:
            mask = np.zeros((h,w), dtype=np.uint8)
        else:
            mask = np.zeros((h,w), dtype=np.uint8)
            records = self.mask_df[self.mask_df['ImageId']==
               image_id]
            for _, r in records.iterrows():
                mask += rle_decode(r['EncodedPixels'], (h,w)).
                   astype(np.uint8)
            mask = np.clip(mask,0,1)
        if self.transform:
            augmented = self.transform(image=image, mask=mask)
            image = augmented['image']
            mask = augmented['mask'].unsqueeze(0).float()
        return image, mask, image_id
```

Listing 2: NucleiDataset simplified

13.1.3 Model creation

```
import segmentation_models_pytorch as smp
model = smp.Unet(
    encoder_name='efficientnet-b3',
    encoder_weights='imagenet',
    in_channels=3,
    classes=1,
    activation=None
).to(DEVICE)
```

Listing 3: Model instantiation using SMP

13.1.4 Loss function

```
bce = nn.BCEWithLogitsLoss()
def dice_loss(preds, targets, eps=1e-7):
    preds = torch.sigmoid(preds)
    preds = preds.view(-1)
    targets = targets.view(-1)
    inter = (preds * targets).sum()
    union = preds.sum() + targets.sum()
    dice = (2. * inter + eps) / (union + eps)
    return 1 - dice

def loss_fn(preds, targets):
    return 0.5 * bce(preds, targets) + 0.5 * dice_loss(preds,
       targets)
```

Listing 4: Hybrid loss: BCE + Dice

13.1.5 Training loop (high-level)

```
for epoch in range(1, EPOCHS+1):
    model.train()
    for images, masks, ids in train_loader:
        images, masks = images.to(DEVICE), masks.to(DEVICE)
        optimizer.zero_grad()
        preds = model(images)
        loss = loss_fn(preds, masks)
        loss.backward()
        optimizer.step()
    # compute epoch metrics, save checkpoints & visuals
```

Listing 5: Training loop outline

Acknowledgements

We thank the Data Science Bowl community for releasing the DSB2018 dataset and the developers of PyTorch, SMP and Albumentations for the tools used in this project.

We are deeply grateful to Dr. Shitala Prasad for his constant encouragement, expert supervision, and for providing us with the opportunity and resources to work on this project.

Authors' Contribution

- **Chaman: Model architecture design**; **training pipeline development and optimization**; full experiment execution; baseline reproduction and comparison (with Suryansh and Yash); implementation of **loss functions, evaluation metrics, and checkpointing**; preparation of inference demo; and major contribution to **report writing, formatting, final editing, and assembly** (with all authors).
- **Suryansh: Dataset acquisition, preprocessing, and RLE mask handling** (with Chaman); design of augmentation and preprocessing pipelines; **visualization generation and analysis**; baseline training and comparative evaluation (with Chaman and Yash); preparation of **presentation slides and poster**; and qualitative result compilation.
- **Yash: Literature review, background theory development, and related work expansion**; design and validation of **evaluation metrics**; interpretation of results and comparative analysis (with Chaman and Suryansh); reproducibility setup including environment configuration; **code structuring and documentation**; and major contributions to **conceptual clarity and manuscript refinement**.

Code and Resource Availability

All resources developed and used in this project are publicly available in our GitHub repository:

https://github.com/chaman-yadav/Identification-and-Segmentation-of-Nuclei-in-Cells

Bibliography

1. Kaggle DSB2018 Dataset: https://www.kaggle.com/c/data-science-bowl-2018
2. Segmentation Models PyTorch (SMP): https://github.com/qubvel/segmentation_models.pytorch
3. EfficientNet Paper: https://arxiv.org/abs/1905.11946
4. MedLiteNet, Lightweight Hybrid Medical Image Segmentation Model: https://arxiv.org/html/2509.03041v1