Lab 02: Semantic Segmentation with UNet and ResNet34_UNet

112550127 Pin-Kuan Chiang

July 22, 2025

1 Implementation Details

1.1 Code Structure

The project is organized into modular files:

- `train.py`: Handles training loop, logging, and checkpointing.
- `evaluate.py`: Performs validation and logs metrics/images to TensorBoard.
- `inference.py`: Loads trained models and evaluates on the test set, plotting loss and Dice score.
- `oxford_pet.py`: Implements dataset loading, preprocessing, and augmentation.
- `models/unet.py`, `models/resnet34_unet.py`: Model architectures.

1.2 `train.py`

The `train.py` script is the main training pipeline. It includes data loading, model construction, training loop, validation evaluation, learning rate scheduling, checkpointing, and TensorBoard logging.

TensorBoard Logging: To monitor training progress, I utilized PyTorch's TensorBoard integration:

```
writer = SummaryWriter(log_dir)
writer.add_scalar('Train/Loss', avg_train_loss, epoch)
writer.add_scalar('Train/Dice_Score', avg_train_dice, epoch)
writer.add_scalar('Learning_Rate', optimizer.param_groups[0]['lr'], epoch)
```

Besides scalar metrics like loss, Dice score, and learning rate, the validation script also logs predicted images and ground truths for visual inspection.

Loss Function: I chose `BCEWithLogitsLoss` because semantic segmentation here is treated as a binary classification task (foreground vs background), and the models output raw logits:

```
criterion = nn.BCEWithLogitsLoss()
```

Optimizer: I used the Adam optimizer for its adaptive learning rate behavior and general good performance in deep learning:

```
optimizer = optim.Adam(model.parameters(), lr=args.learning_rate)
```

Learning Rate Scheduler: The scheduler is configurable. Two main schedulers were experimented:

- **CosineAnnealingLR** gradually decreases the learning rate in a cosine fashion. This helps slow down learning near the end of training to reduce overfitting. It slightly improved stability in later epochs.
- **ReduceLROnPlateau** monitors validation loss and reduces learning rate if no improvement is seen. This was inspired by its use in the original ResNet paper.

```
if args.scheduler == 'cosine':
    scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=args.epochs)
elif args.scheduler == 'plateau':
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', ...)
```

Training Loop and Evaluation: Each epoch performs standard forward and backward passes, computes Dice score, logs results, and calls `evaluate()` on the validation set:

```
avg_val_loss, avg_val_dice = evaluate(model, val_loader, device, writer, epoch, "val")
```

The validation metrics are also logged to TensorBoard.

TensorBoard Summary: During training, the following items are recorded:

- Training loss and Dice score per epoch
- Validation loss and Dice score per epoch
- Learning rate
- Model computation graph (via `add_graph`)
- Visual samples: predicted masks vs. ground truth

Arguments and Augmentation: The script supports multiple command-line arguments to control training:

```
--model unet or resnet34_unet
--scheduler cosine or plateau
--batch_size, --learning_rate, --epochs, etc.
--augment / --no-augment
```

Data augmentation is enabled by default during training and will be elaborated in the data preprocessing section.

1.3 `evaluate.py`

This script performs model evaluation on the validation or test set. It calculates two metrics—binary cross-entropy loss and Dice score—without updating model weights. Evaluation is done under `torch.no_grad()` and with `net.eval()` to ensure proper inference behavior.

```
criterion = nn.BCEWithLogitsLoss()
with torch.no_grad():
    for batch in data_loader:
        outputs = net(images)
        loss = criterion(outputs, masks)
        dice = dice_score(outputs, masks)
```

TensorBoard Visualization: For each epoch, the first batch from the validation set is visualized in TensorBoard, including:

- Original image (denormalized from ImageNet format)
- Ground truth segmentation mask
- Predicted mask (as probability map via sigmoid)
- Binarized prediction (threshold at 0.5)

To properly display the input images, they are denormalized using the inverse of ImageNet statistics:

```
mean = torch.tensor([0.485, 0.456, 0.406]).view(1,3,1,1)
std = torch.tensor([0.229, 0.224, 0.225]).view(1,3,1,1)
denormalized_images = (images * std) + mean
```

This allows visual comparison between input, ground truth, and prediction to better assess model behavior across epochs.

Note: The script also logs per-epoch validation loss and Dice score to TensorBoard under the tag `val/` (or `test/` depending on phase), making it easy to track model generalization over time.

1.4 `inference.py`

This script performs final evaluation on the test set using a trained model checkpoint. Its structure and functionality closely resemble `evaluate.py`, but it is specifically designed for inference and testing after training is complete.

Dataset Mode: Unlike validation, the dataset is loaded in `test` mode:

```
test_dataset = SimpleOxfordPetDataset(root=args.data_path, mode="test")
```

Model Loading: The model is instantiated based on the selected architecture and then loaded from a checkpoint file:

```
model = load_model(args.model, args.model_type, device)
```

TensorBoard Logging: Like the validation phase, the script logs inference results to TensorBoard. Logged items include:

- Per-batch test loss and Dice score
- Average loss and Dice score over the test set
- Standard deviation of loss and Dice score
- Visual samples of input images (denormalized), ground truth, predicted probability masks, and binary predictions

Evaluation Loop: Metrics are accumulated over the entire test set:

```
for batch in test_loader:
    outputs = model(images)
    loss = criterion(outputs, masks)
    dice = dice_score(outputs, masks)
```

Visualization: As with `evaluate.py`, the first batch is visualized in TensorBoard after denormalization, enabling qualitative inspection of predictions.

Summary: After testing, the script prints overall average loss, average Dice score, and their standard deviations. These help quantify both model performance and prediction consistency on unseen data.

1.5 NN Architecture

1.5.1 `unet.py`

The following implementation provides a modular and readable realization of the U-Net architecture in PyTorch. It includes several submodules to structure the encoder, decoder, and intermediate connections.

DoubleConv Module The `DoubleConv` module is designed to perform two consecutive convolutional operations, each followed by batch normalization and a ReLU activation. This block is a core unit of U-Net.

```
class DoubleConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(DoubleConv, self).__init__()
        self.double_conv = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)
        )
```

Padding is set to 1 to maintain spatial dimensions and avoid feature size shrinkage. Batch normalization is added to stabilize training and prevent exploding values due to ReLU, which does not inherently bound output magnitude.

Down Module The `Down` class performs spatial downsampling using a max-pooling layer, followed by a `DoubleConv` block.

```
class Down(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Down, self).__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, out_channels)
        )
```

This follows the typical U-Net encoder structure: pooling for resolution reduction and convolution for feature enrichment.

Up Module and Cropping The `Up` module performs upsampling with a transposed convolution, followed by concatenation with the corresponding encoder feature map (skip connection), and a `DoubleConv` block.

```
class Up(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Up, self).__init__()
        self.up = nn.ConvTranspose2d(in_channels, in_channels // 2, kernel_size=2, stride=2)
        self.conv = DoubleConv(in_channels, out_channels)
```

In standard U-Net implementations, spatial mismatches between encoder and decoder feature maps may arise due to rounding in pooling operations. Therefore, a custom `cropping` function is often used to align dimensions before concatenation:

```
def cropping(tensor, target_tensor):
    _, _, h, w = target_tensor.shape
    tensor_center_h = tensor.shape[2] // 2
    tensor_center_w = tensor.shape[3] // 2
    start_h = tensor_center_h - h // 2
    start_w = tensor_center_w - w // 2
    return tensor[:, :, start_h:start_h + h, start_w:start_w + w]
```

However, in our implementation, since `DoubleConv` includes padding to maintain spatial dimensions, the cropping step is no longer strictly necessary. Nevertheless, I retained the cropping function for clarity and compatibility with conventional U-Net code structures.

OutConv Module The final layer maps the features to the number of output classes using a 1×1 convolution:

```
class OutConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(OutConv, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=1)
```

UNet Class The `UNet` class assembles all the components:

```
class UNet(nn.Module):
    def __init__(self, n_channels, n_classes):
        super(UNet, self).__init__()
        self.inc = DoubleConv(n_channels, 64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128, 256)
        self.down3 = Down(256, 512)
        self.down4 = Down(512, 1024)
        self.up1 = Up(1024, 512)
        self.up2 = Up(512, 256)
        self.up3 = Up(256, 128)
        self.up4 = Up(128, 64)
        self.outc = OutConv(64, 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
```

The network follows a symmetrical encoder-decoder structure with skip connections between matching levels. Each layer is clearly modularized, following the original U-Net design, with minor adjustments such as explicit cropping to ensure dimensional alignment.

This implementation provides a clear and extensible foundation for segmentation tasks, adhering closely to the spirit of the original U-Net while integrating practical improvements like batch normalization and manual cropping.

1.5.2 `resnet34_unet.py`

This section presents a hybrid U-Net architecture based on a ResNet34 encoder. The key idea is to leverage a powerful residual network for feature extraction while using a decoder inspired by U-Net for semantic segmentation.

BasicBlock and Shortcut Design The ResNet34 encoder is composed of residual blocks known as `BasicBlock`. Each `BasicBlock` consists of two 3×3 convolutional layers with Batch Normalization and ReLU activation:

```
class BasicBlock(nn.Module):
    def __init__(self, inplanes, planes, stride=1, downsample=None):
        ...
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
```

A *shortcut* connection is used to add the input directly to the output, either as identity or using a 1×1 convolution (when reducing channels or changing resolution). This design improves gradient flow and training stability in deep networks.

Encoder Design The encoder in `ResNet34UNet` follows the structure of the original ResNet34. It processes the input through the initial convolution, max pooling, and a series of residual layers:

```
class ResNet34Encoder(nn.Module):
    def __init__(self, in_channels):
        ...
        self.layer1 = self._make_layer(BasicBlock, 64, 3)
        self.layer2 = self._make_layer(BasicBlock, 128, 4, stride=2)
        self.layer3 = self._make_layer(BasicBlock, 256, 6, stride=2)
        self.layer4 = self._make_layer(BasicBlock, 512, 3, stride=2)
```

During the forward pass, it returns five feature maps from different stages of the encoder for use in skip connections:

```
def forward(self, x):
    x1 = self.relu(self.bn1(self.conv1(x)))
    x2 = self.layer1(x1)
    x3 = self.layer2(x2)
    x4 = self.layer3(x3)
    x5 = self.layer4(x4)
    return x1, x2, x3, x4, x5
```

Decoder Design The `UNetDecoder` implements the decoding path with upsampling and skip connections. Unlike the original U-Net, this design avoids manual cropping by using padded convolutions to ensure that all feature maps have matching spatial sizes.

```
class UNetDecoder(nn.Module):
    def __init__(self, num_classes=3):
        ...
        self.up1 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
        self.conv1 = self._double_conv(512, 256)

        self.up2 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.conv2 = self._double_conv(256, 128)

        self.up3 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.conv3 = self._double_conv(128, 64)

        self.up4 = nn.ConvTranspose2d(64, 64, kernel_size=2, stride=2)
        self.conv4 = self._double_conv(128, 64)

        self.final_up = nn.ConvTranspose2d(64, 64, kernel_size=2, stride=2)
        self.final_conv = self._double_conv(64, 64)

        self.out_conv = nn.Conv2d(64, num_classes, kernel_size=1)
```

Each stage performs upsampling followed by concatenation with the corresponding encoder feature map. Since all spatial sizes match, no cropping is needed:

```
x = self.up1(x5)
x = torch.cat([x, x4], dim=1)
x = self.conv1(x)
...
x = self.final_up(x)
x = self.final_conv(x)
x = self.out_conv(x)
```

Complete Model: ResNet34UNet The full model is built by combining the ResNet34-based encoder with the U-Net-style decoder. The encoder extracts multiscale features while the decoder performs reconstruction with skip connections to preserve spatial detail:

```
class ResNet34UNet(nn.Module):
    def __init__(self, in_channels=3, num_classes=3):
        super(ResNet34UNet, self).__init__()
        self.encoder = ResNet34Encoder(in_channels)
        self.decoder = UNetDecoder(num_classes)

    def forward(self, x):
        features = self.encoder(x)
        output = self.decoder(features)
        return output
```

This hybrid architecture combines the representational strength of ResNet with the fine localization ability of U-Net, making it suitable for tasks such as semantic segmentation and medical imaging.

2 Data Preprocessing

Initially, no preprocessing was applied to the dataset. However, during training, the model quickly overfit to the training data and failed to generalize. After reading discussions on e3 forums, data augmentation was added using `torchvision.transforms` to improve generalization. Furthermore, by referring to the documentation of `torchvision.models`, it became clear that pretrained models assume ImageNet-style normalization, so that step was also incorporated.

2.1 Normalization

All input images and masks are resized to 256×256. The normalization process ensures compatibility with pretrained backbones (e.g., ResNet34) trained on ImageNet. It consists of:

- Converting image to `float32` and normalizing to $[0, 1]$.
- Standardizing pixel values using ImageNet statistics:

  - Mean = $[0.485, 0.456, 0.406]$
  - Std = $[0.229, 0.224, 0.225]$
- Converting images from HWC (height, width, channel) to CHW format for PyTorch.

```
# Resize
image = np.array(Image.fromarray(sample["image"]).resize((256, 256), Image.BILINEAR))
mask = np.array(Image.fromarray(sample["mask"]).resize((256, 256), Image.NEAREST))

# Normalize to [0, 1]
image = image.astype(np.float32) / 255.0

# ImageNet normalization
imagenet_mean = np.array([0.485, 0.456, 0.406], dtype=np.float32)
imagenet_std = np.array([0.229, 0.224, 0.225], dtype=np.float32)
image = (image - imagenet_mean) / imagenet_std

# Format conversion
sample["image"] = np.moveaxis(image, -1, 0)
sample["mask"] = np.expand_dims(mask, 0)
```

2.2 Augmentation (Training Only)

To mitigate overfitting and improve robustness, data augmentation is applied only during training. The augmentation is implemented via `torchvision.transforms.functional` and includes a variety of spatial and photometric transformations, all applied with probabilistic triggers:

- **Random Horizontal Flip** (50%): helps capture left-right symmetry.
- **Random Vertical Flip** (20%): useful for vertical invariance.
- **Random Rotation** ($\pm 15°$, 30%): provides orientation robustness.
- **Color Jitter** (40%): adjusts brightness, contrast, saturation, and hue.
- **Gaussian Blur** (20%): simulates soft focus or sensor blur.
- **Random Affine Transform** (25%): applies scale, shear, and translation variations.

Example code:

```
if random.random() > 0.5:
    image = TF.hflip(image)
    mask = TF.hflip(mask)

if random.random() > 0.8:
    image = TF.vflip(image)
    mask = TF.vflip(mask)

if random.random() > 0.7:
    angle = random.uniform(-15, 15)
    image = TF.rotate(image, angle)
    mask = TF.rotate(mask, angle)

if random.random() > 0.6:
    image = color_jitter(image)

if random.random() > 0.8:
    image = TF.gaussian_blur(image, kernel_size=3, sigma=0.5)

if random.random() > 0.75:
    image = TF.affine(image, angle=0, translate=(5,5), scale=1.05, shear=2)
    mask = TF.affine(mask, angle=0, translate=(5,5), scale=1.05, shear=2)
```

Because all augmentations are implemented with padding-aware transforms and kept spatially consistent between images and masks, no cropping is needed after augmentation. This pipeline results in a more robust model and improved generalization across unseen data.

3 Experiment Results

This section presents and analyzes the results of various experiments across three aspects: learning rate (LR), batch size (BS), and data augmentation (AUG). For each, I compare performance using training loss, validation loss, and Dice score. Screenshots from TensorBoard visualizations are provided to illustrate training dynamics.

Due to hardware limitations—specifically, a GPU with only 6GB of VRAM—the maximum feasible batch size was constrained to 8. This limitation influenced experimental design, as smaller batch sizes tend to yield noisier gradient estimates, which can impact convergence stability and final performance. Despite this, meaningful trends were still observable across experiments.

3.1 Learning Rate (LR)

Condition: Batch size = 4, no augmentation, epochs = 50

Observation: LR too large (e.g., 0.001) may overfit. The sweet spot is around $1e^{-4}$.

Model	Train Loss	Train Dice	Val Loss	Val Dice
ResNet34_UNet (1e-3)	0.0648	0.9668	0.2560	0.8928
ResNet34_UNet (1e-4)	0.0549	0.9712	0.2345	0.9216
ResNet34_UNet (1e-5)	0.0435	0.9775	0.4403	0.8650
UNet (1e-3)	0.0500	0.9749	0.5767	0.8485
UNet (1e-4)	0.0174	0.9908	0.5536	0.9012
UNet (1e-5)	0.0454	0.9763	0.4146	0.8919

Table 1: Comparison of learning rates for both models (no augmentation, bs=4, 50 epochs)

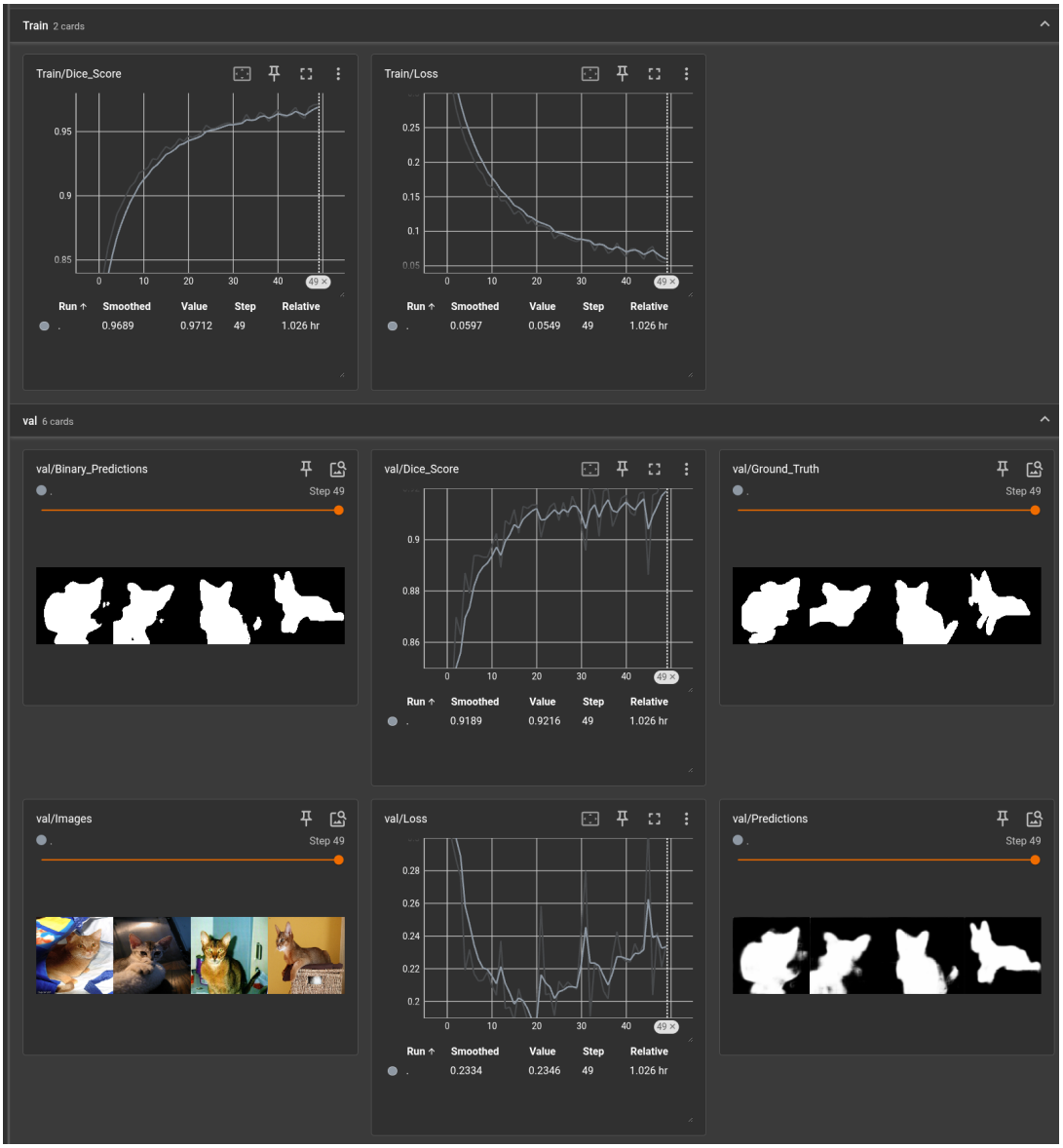

Figure 1: Best LR result (ResNet34_UNet, lr=1e-4)

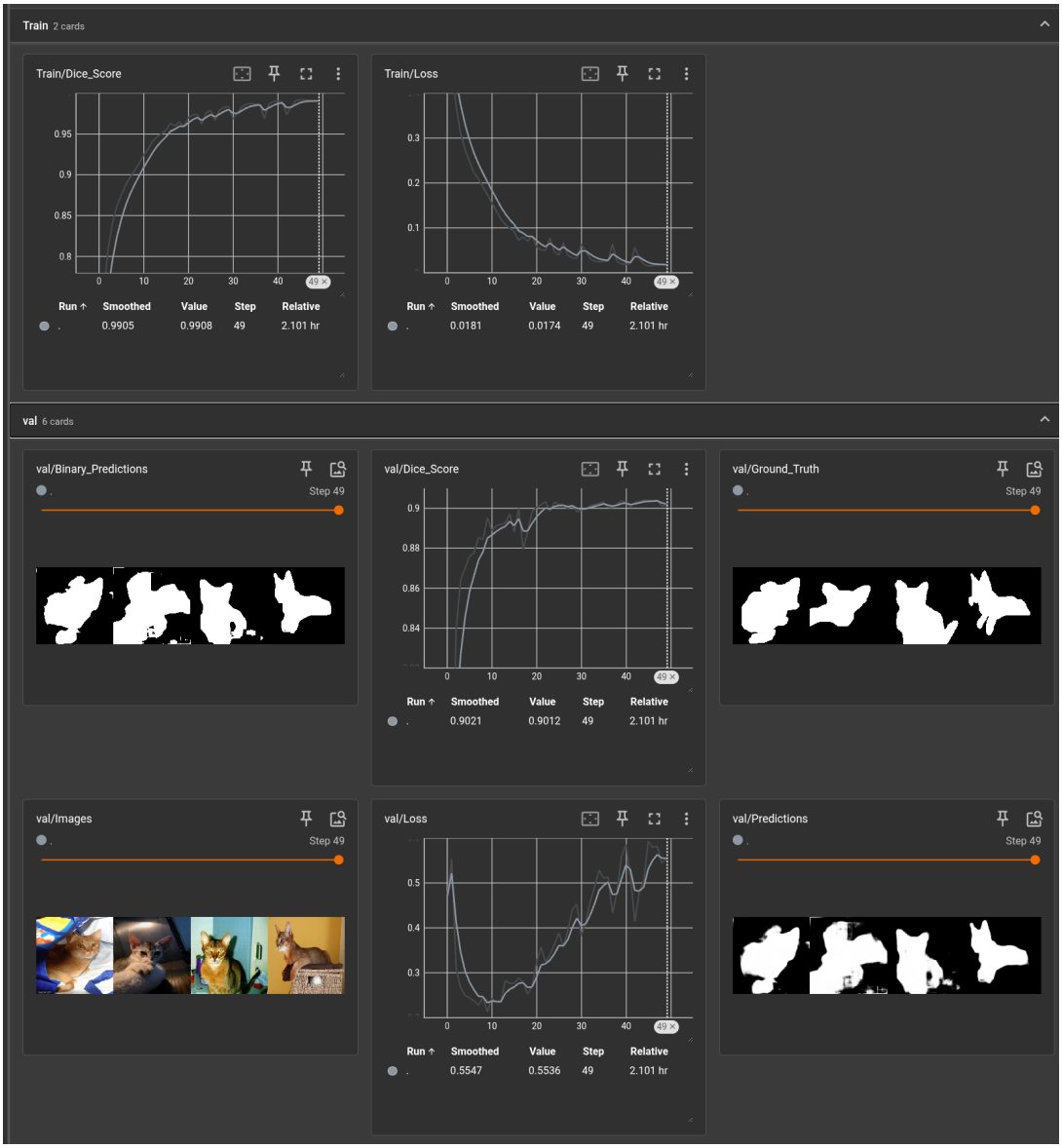

Figure 2: Best LR result (UNet, lr=1e-4)

3.2 Batch Size (BS)

Condition: LR = $1e^{-5}$, no augmentation, epochs = 50

Observation: Increasing batch size to 8 with a cosine scheduler improves validation performance. For ResNet34, I also tried the original paper's setup using lr=0.1 with ReduceLROnPlateau, which resulted in very poor convergence.

Model	Train Loss	Train Dice	Val Loss	Val Dice
ResNet34_UNet (bs=4)	0.0435	0.9775	0.4403	0.8650
ResNet34_UNet (bs=8)	0.0584	0.9745	0.3504	0.8626
UNet (bs=4)	0.0454	0.9763	0.4146	0.8919
UNet (bs=8)	0.0514	0.9791	0.2800	0.8884

Table 2: Comparison of batch size with and without cosine scheduler (lr=1e-5)

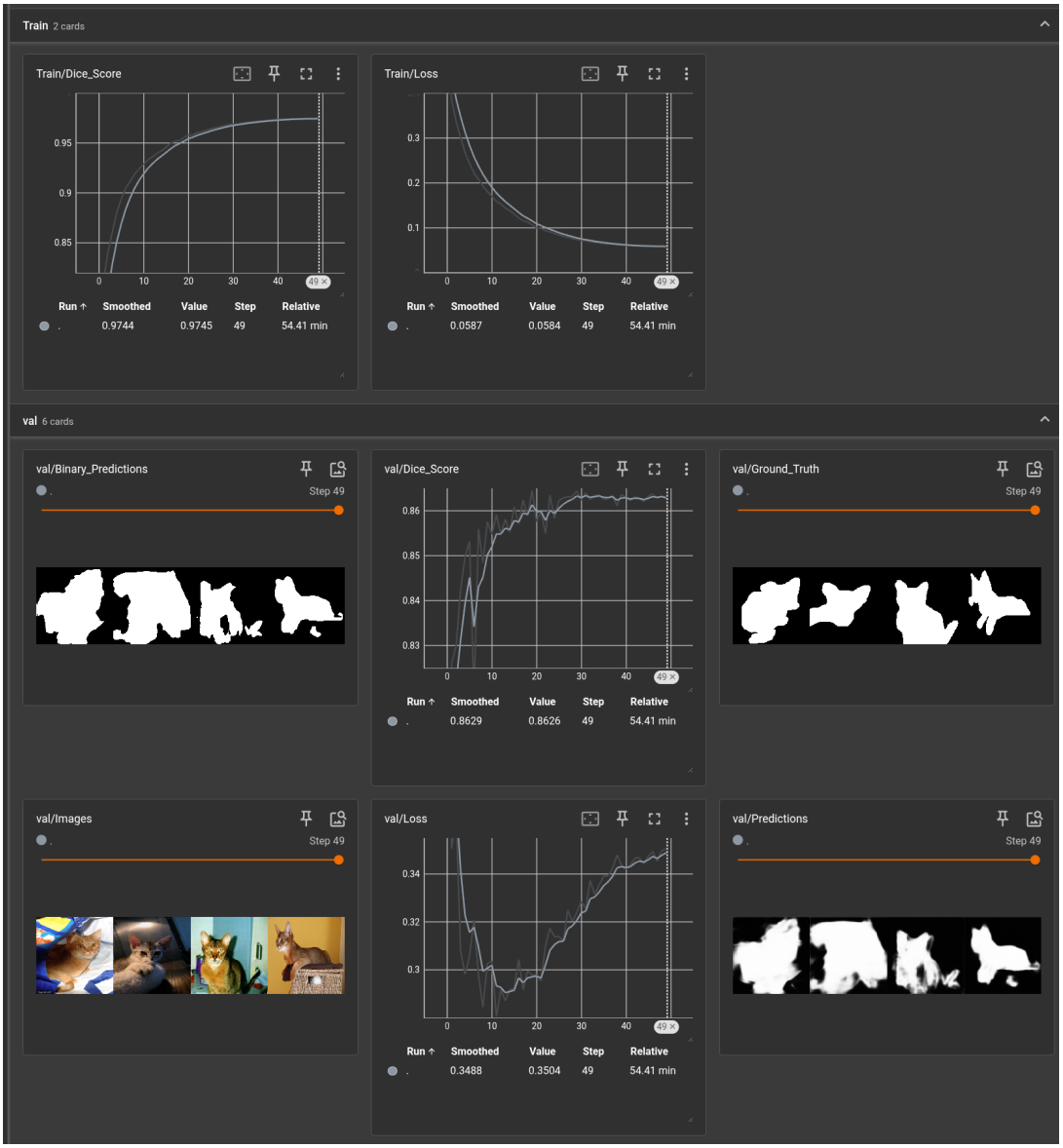

Figure 3: Best BS result (ResNet34_Unet, bs=8, lr=1e-5)

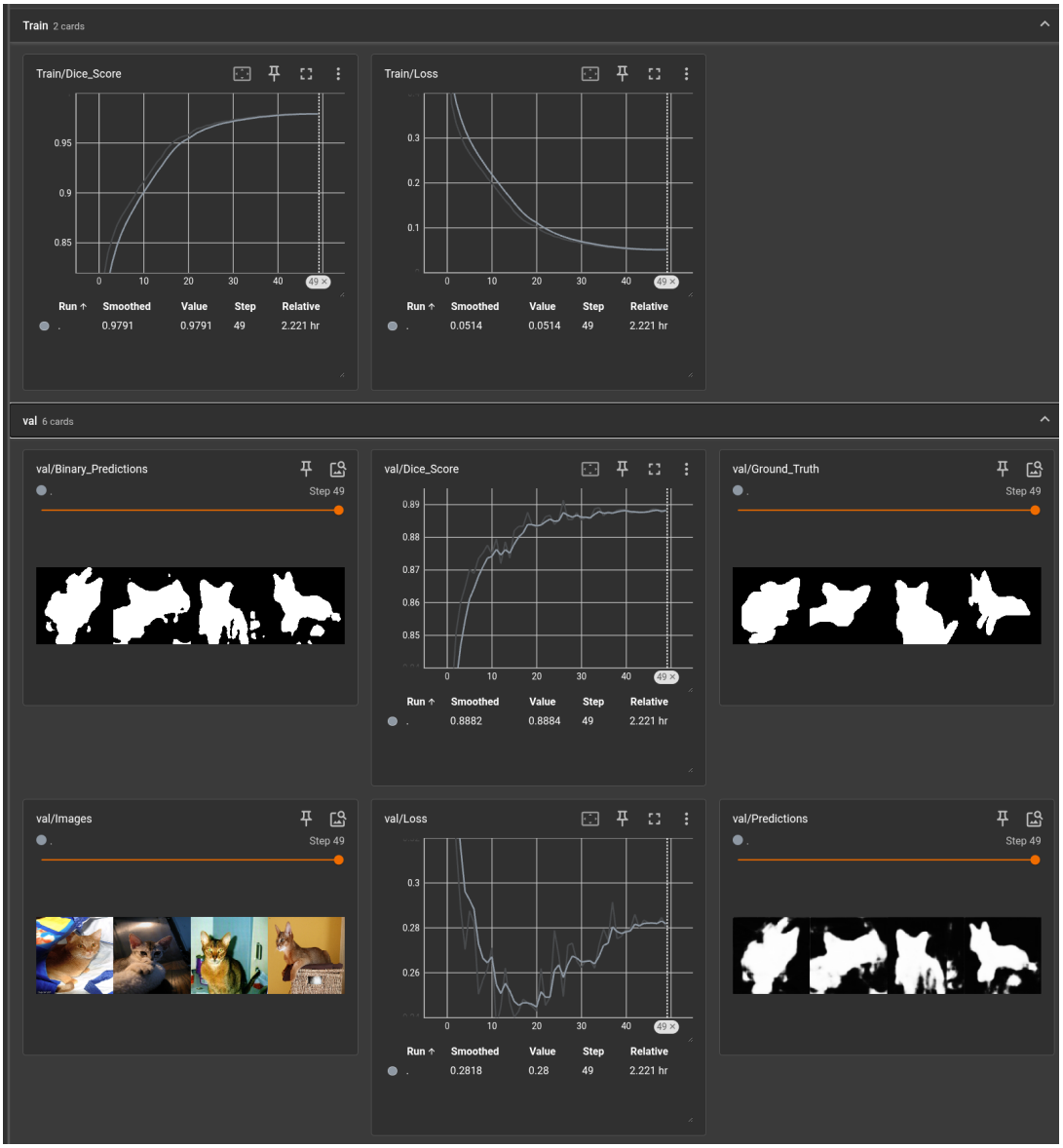

Figure 4: Best BS result (UNet, bs=8, lr=1e-5)

3.3 Data Augmentation (AUG)

Condition: Batch size = 8, LR = $1e^{-4}$, epochs = 50, using cosine scheduler

Observation: Augmentation significantly reduces overfitting. The gap between training and validation loss shrinks. Dice score improves for both models, especially on validation.

Model	Train Loss	Train Dice	Val Loss	Val Dice
ResNet34_UNet (no aug)	0.0549	0.9712	0.2345	0.9216
ResNet34_UNet (aug)	0.1084	0.9467	0.1448	0.9320
UNet (no aug)	0.0174	0.9908	0.5536	0.9012
UNet (aug)	0.0952	0.9531	0.1294	0.9385

Table 3: Effect of data augmentation on generalization (bs=8, lr=1e-4)

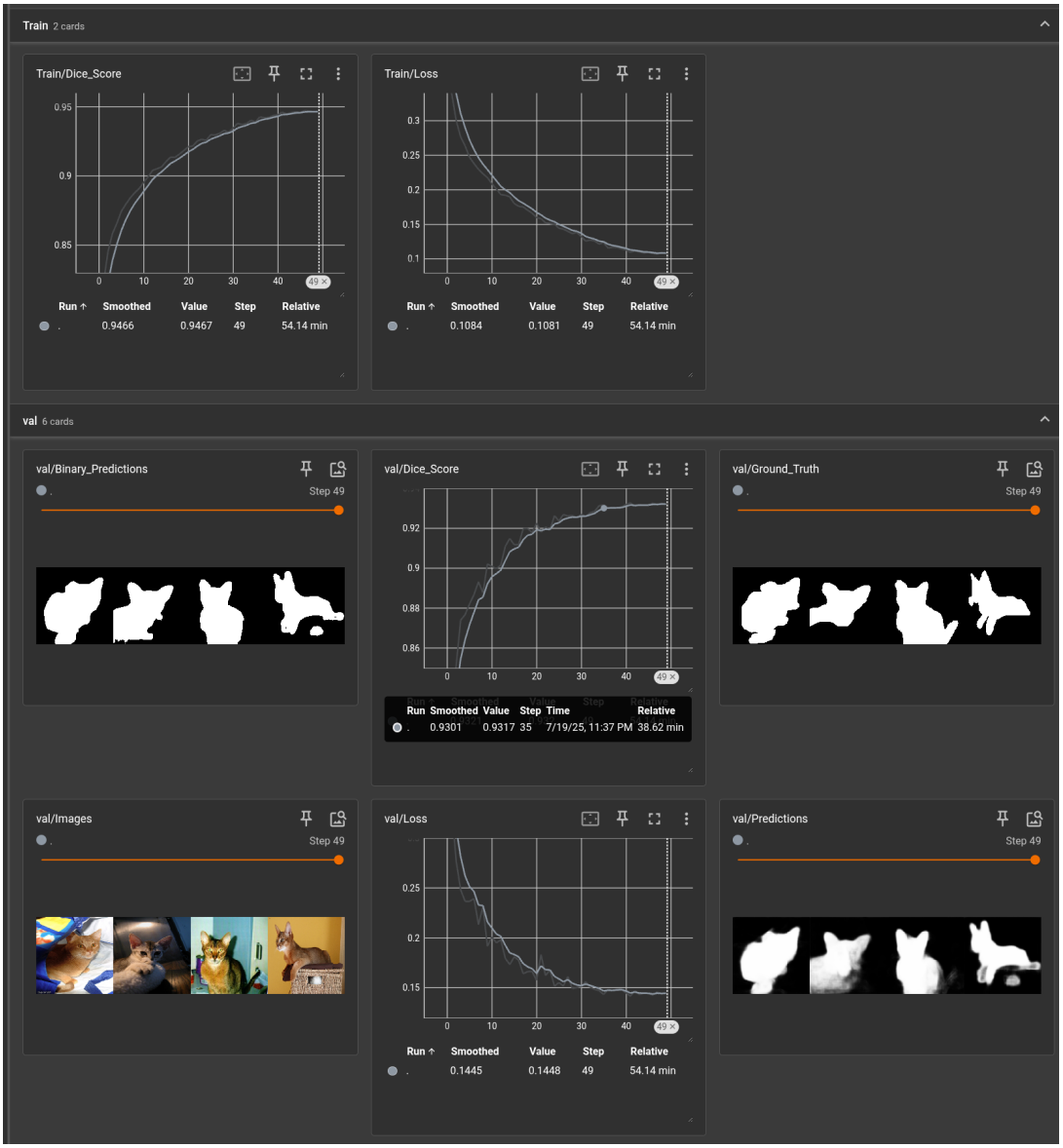

Figure 5: Best AUG result (ResNet34_Unet, augmentation with cosine scheduler)

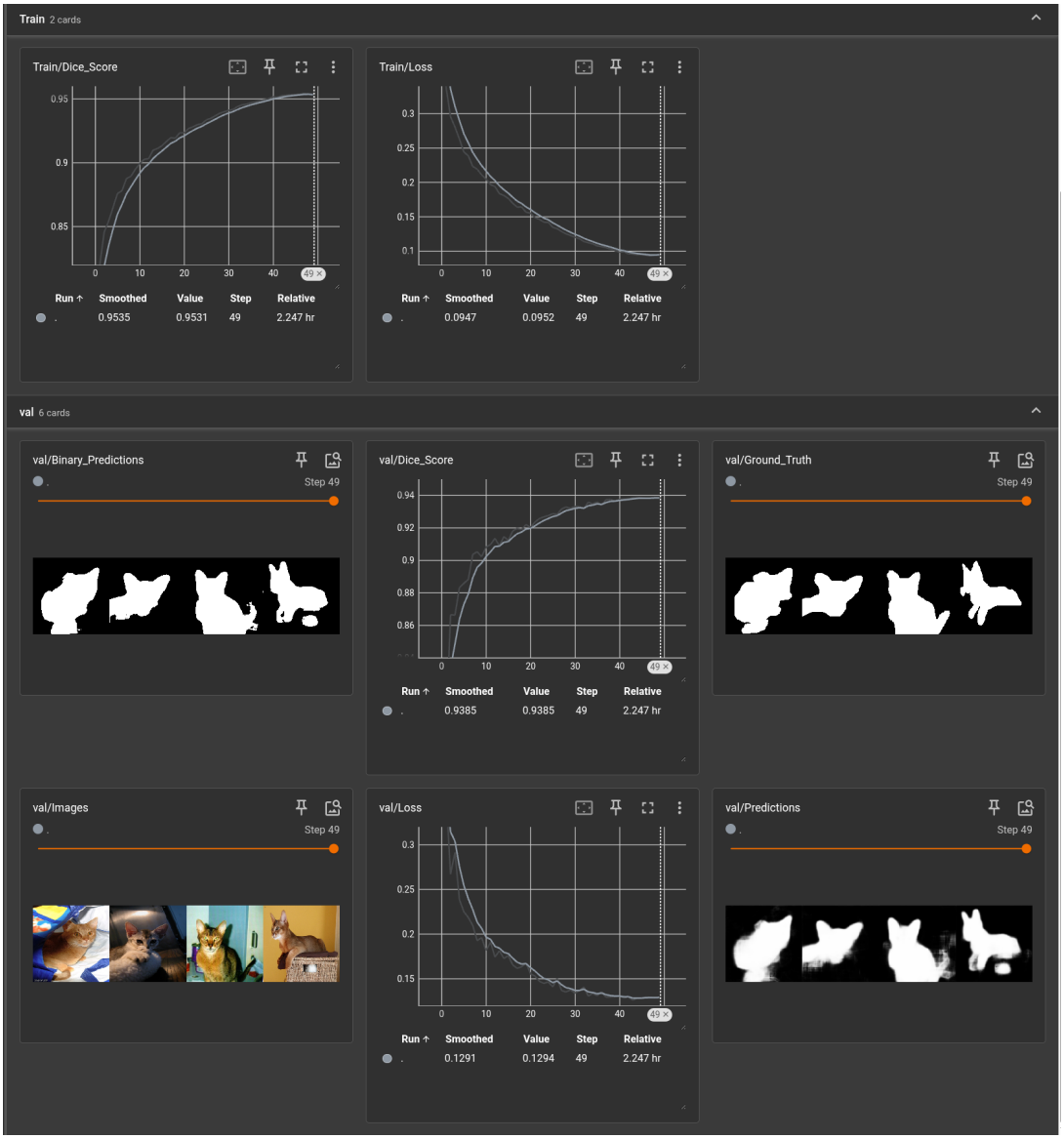

Figure 6: Best AUG result (UNet, augmentation with cosine scheduler)

3.4 Best Results

The best performance for both models was achieved under the same experimental setting:

- Batch size = 8
- Learning rate = 1×10^{-4}
- Optimizer: Adam
- Scheduler: Cosine annealing
- Data augmentation: Enabled
- Epochs = 50

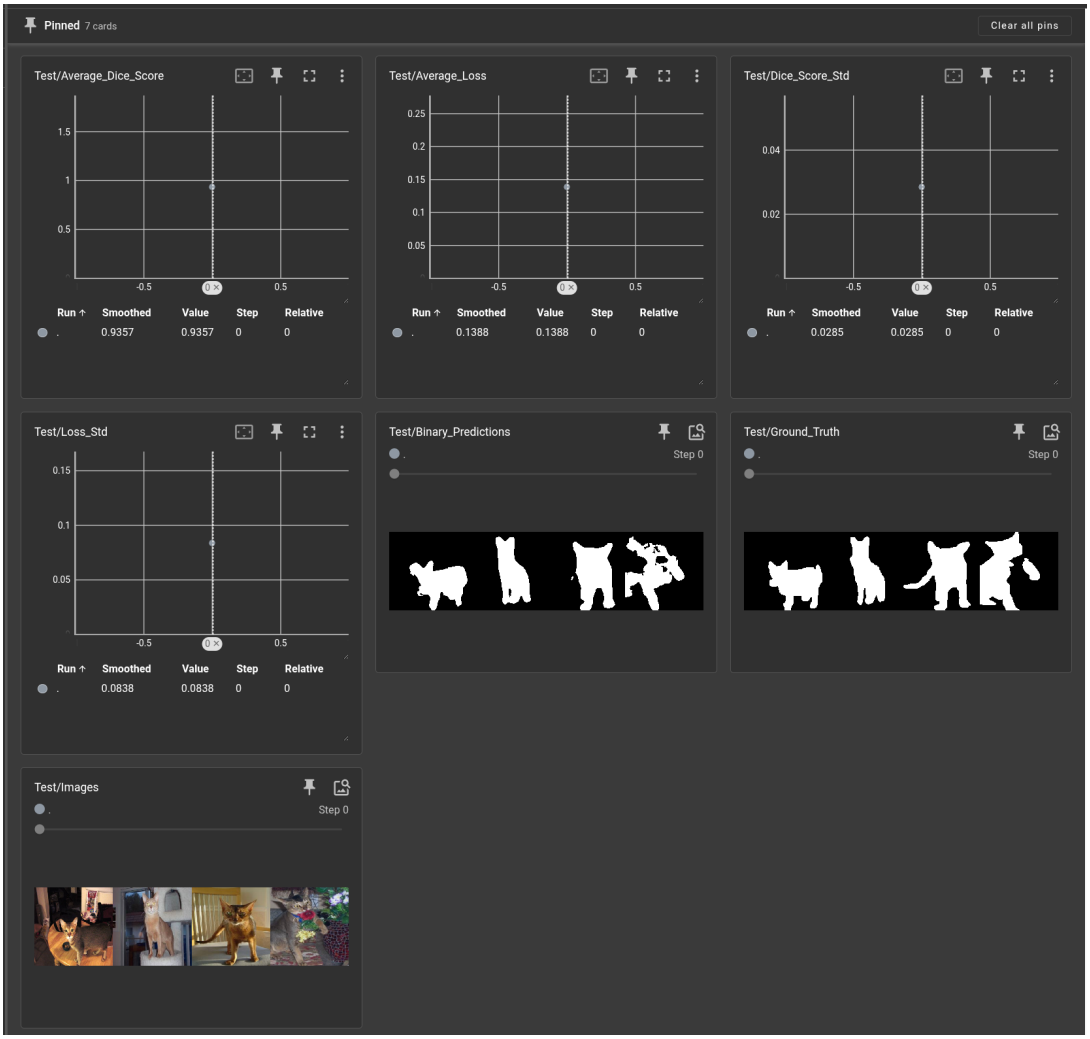

Figure 7: Best testing (ResNet34_Unet)

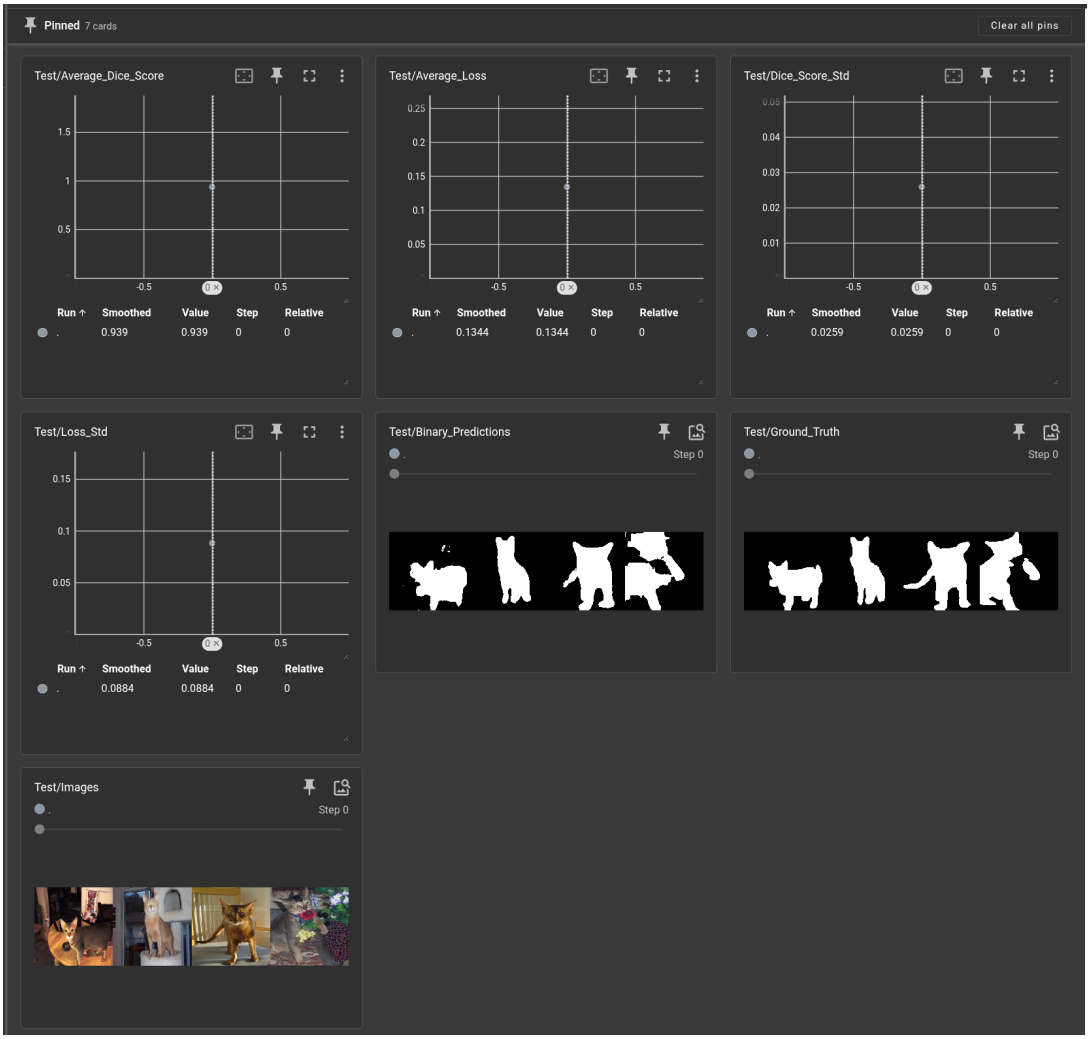

Figure 8: Best testing result (UNet)

Metric	ResNet34_UNet	UNet
Average Validation Loss	0.1388	0.1344
Average Dice Score	**0.9357**	**0.9390**
Validation Loss Std	0.0838	0.0884
Dice Score Std	0.0285	0.0259

Table 4: Best results for each model with bs=8, lr=1e-4, cosine scheduler, and augmentation

Although both models performed well, UNet slightly outperformed ResNet34_UNet in both average loss and Dice score. The results indicate that simple U-Net architectures, when paired with proper regularization and augmentation, can be highly effective on this task. The standard deviation values are low for both models, demonstrating stable performance across validation batches.

4 Execution Steps

All commands must be executed from the root directory of the project. Below are the instructions to train the best models and perform inference using the saved checkpoints.

4.1 Training Best Models

The following commands train the best-performing U-Net and ResNet34-UNet models using data augmentation, cosine learning rate scheduler, batch size of 8, and a learning rate of 1×10^{-4} for 50 epochs.

```
# Train ResNet34-UNet
python ./src/train.py --model resnet34_unet -lr 1e-4 --scheduler cosine --augment

# Train U-Net
python ./src/train.py --model unet -lr 1e-4 --scheduler cosine --augment
```

These commands will save model checkpoints under the `./saved_models/` directory after training completes.

4.2 Inference Using Trained Models

Once training is finished, the following commands perform inference using the best checkpoints. The model path must point to the corresponding `.pth` file, and the model type must be specified.

```
# Inference using ResNet34-UNet
python ./src/inference.py --model saved_models/checkpoint_resnet34_unet_ep50_lr0.0001
    _bs8_cosine.pth --model_type resnet34_unet

# Inference using U-Net
python ./src/inference.py --model saved_models/checkpoint_unet_ep50_lr0.0001_bs8_cosine.pth --
    model_type unet
```

These scripts will generate predictions for the test set and save the results under the `./runs`, the directory by default. Result will also be showed in terminal. Ensure all required packages are installed and the environment is activated prior to execution.

5 Discussion

5.1 Alternative Architectures

Two additional architectures that have proven effective for binary segmentation tasks are SegNet and U^2-Net:

- **SegNet** SegNet adopts a symmetric encoder–decoder structure based on VGG16. In the encoder, each convolutional block comprises two or three conv–ReLU layers followed by max-pooling. Crucially, SegNet stores the pooling indices (locations of maximum activations) and reuses them in the decoder to perform non-learned upsampling via "max-unpooling." This approach:
  - Eliminates the need for learnable upsampling filters, reducing parameter count by up to 50% compared to fully-convolutional decoders.
  - Precisely restores feature map topology, preserving boundary sharpness without additional post-processing.
  - Lowers memory footprint at inference, making it suitable for real-time or embedded deployment.

  Empirical studies on road-scene and biomedical benchmarks show SegNet achieving competitive IoU with faster inference speeds than heavier decoders [1].

- **U^2-Net** U^2-Net introduces "Residual U-blocks" (RSUs): each RSU embeds a miniaturized U-Net within a single stage. The full network thus forms a two-level nested U-structure:
  - **RSU Block**: Consists of a series of down-sampling conv layers and corresponding up-sampling deconv layers, with local skip connections, enabling multi-scale feature extraction within one block.
  - **Nested Hierarchy**: Stacking RSUs in both encoder and decoder yields deeper representational capacity without quadratic growth in parameters.
  - **Edge Preservation**: The dense skip connections across scales ensure fine structure retention, crucial for delineating small or thin objects.

  On salient object segmentation tasks, U^2-Net surpasses many pre-trained backbones while training from scratch on moderate-sized datasets, demonstrating robustness and high mask quality [2].

5.2 Research Directions

- **Enhanced Preprocessing for Robustness** Building a richer preprocessing pipeline can expose the model to diverse visual conditions before learning:
  - *Adaptive Histogram Equalization*: Locally adjusts contrast to highlight faint structures in low-contrast images.
  - *Illumination Normalization*: Removes lighting artifacts via Retinex or homomorphic filtering, reducing domain shift between indoor/outdoor captures.
  - *Edge-Preserving Smoothing*: Filters such as bilateral or guided filters erase noise while keeping boundaries crisp, simplifying the segmentation task.

  These techniques can expand the effective training distribution without collecting new data, improving generalization to unseen environments.

- **Overfitting Mitigation Strategies** When models memorize training patterns, performance on new images suffers. To counteract:
  - *Stronger Regularization*: Increase weight decay, apply spatial dropout in intermediate layers, or use label smoothing to discourage overconfident predictions.
  - *Early Stopping with Patience*: Monitor validation IoU and halt training when no improvement occurs for several epochs to avoid degradation.

  Together, these measures reduce model variance and lead to more reliable segmentation across datasets.

5.3 Reflections

- **Data Augmentation Insufficiency** Simple flips and crops may be inadequate for real-world variability, causing overfitting. More diverse augmentations to consider:
  - *Elastic Deformations*: Simulate non-rigid deformations common in biological tissues.
  - *Random Occlusions*: CutMix or CutOut randomly mask out regions to teach the network robustness to missing information.
  - *Style Transfer*: Apply neural style transforms to mimic different weather or sensor characteristics.

- **Schedulers and Optimizers** Optimal learning dynamics hinge on scheduler and optimizer choice:
  - *Cosine Annealing / Cyclic LR*: Periodically vary learning rate to escape local minima and improve final convergence.
  - *Warm-Restart Schedules*: Combine with cosine annealing to reset learning rate, encouraging exploration.
  - *Adaptive Optimizers*: AdamW mitigates weight decay issues; Ranger (Rectified Adam + Lookahead) provides smoother loss landscapes.

- **Implementation Simplicity** Complex cropping/resizing pipelines introduce coordinate-mapping errors and extra code paths. Using *padding* instead:
  - Maintains original image alignment and dimensions throughout the network.
  - Simplifies mask-to-image correspondence, easing loss computation and metric evaluation.
  - Reduces preprocessing code complexity, lowering the barrier for replication and extension.

6 Demo Link

https://youtu.be/K1NLZs1u2nA?si=TdTcPk59y3Bf0CZY

References

[1] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[2] X. Qin, Z. Zhang, C. Huang, M. Dehghan, O. R. Zaiane, and M. Jagersand, "U^2-net: Going deeper with nested u-structure for salient object detection," *Pattern Recognition*, vol. 106, p. 107404, 2020.

LM Usage

Language was used during development to assist with the implementation of more complex components of the codebase, such as custom modules and tensor dimension handling. While Copilot provided helpful code suggestions that accelerated prototyping, many of its outputs contained subtle bugs or incorrect logic. These issues were identified through careful manual review and corrected before integration.

In addition, language models were also used to assist in drafting and polishing English content for the project presentation. This helped ensure that slides were clearly phrased and free of common grammatical inconsistencies, improving overall communication quality.