Artificial Vision (VIAR25/26)

Lab 5: Advanced Image Segmentation

Prof. David Olivieri

UVigo

October 2, 2025

Lab Objectives

Today's Mission

Implement semantic segmentation architectures from scratch, master advanced loss functions, and build a trainable Mini-SAM model for prompt-based segmentation.

Learning Outcomes

1. **Build** FCN-8s from pretrained ResNet with skip connections
2. **Implement** DeepLabV3+ with ASPP module and atrous convolutions
3. **Master** advanced loss functions: Dice, Focal, combined strategies
4. **Train** Mini-SAM from scratch with point/box prompting
5. **Analyze** architectural trade-offs and performance metrics

Datasets

- **PASCAL VOC 2012**: 21 classes, semantic segmentation benchmark
- **VOC Subset**: 500 images for Mini-SAM training
- **Synthetic shapes**: Quick testing and debugging

Lab Structure

Part 1: FCN Architecture (40 min)

- **Task 1.1**: Convert ResNet to fully convolutional network
- **Task 1.2**: Implement score layers and progressive upsampling
- **Task 1.3**: Build FCN-32s, FCN-16s, FCN-8s with skip connections
- **Task 1.4**: Train and compare skip connection impact

Part 2: DeepLabV3+ with ASPP (40 min)

- **Task 2.1**: Implement atrous convolution operations
- **Task 2.2**: Build complete ASPP module with 5 branches
- **Task 2.3**: Create encoder-decoder with low-level feature fusion
- **Task 2.4**: Compare performance with FCN variants

Part 3: Mini-SAM Training from Scratch (40 min)

- **Task 3.1**: Build lightweight Mini-SAM architecture
- **Task 3.2**: Implement prompt encoding (points and boxes)
- **Task 3.3**: Train Mini-SAM with simulated prompts
- **Task 3.4**: Comparative analysis of all methods

Part 1: Fully Convolutional Networks (FCN)

From Classification to Dense Prediction

Transform a pretrained ResNet into a segmentation network by replacing fully connected layers with 1×1 convolutions and adding skip connections to recover spatial detail.

FCN Architecture Variants

- **FCN-32s**: Single $32\times$ upsampling from conv5
- **FCN-16s**: Add skip from conv4 (stride 16)
- **FCN-8s**: Add skip from conv3 (stride 8)
- Progressive improvement in boundary detail

Key Mathematical Insight

FC layer as convolution:

$$\mathbf{W}_{\text{FC}} \in \mathbb{R}^{K \times D} \equiv \mathbf{W}_{\text{conv}} \in \mathbb{R}^{K \times H' \times W' \times C}$$

Enables arbitrary input sizes!

Implementation Components

1. Load pretrained ResNet50
2. Extract features at multiple scales
3. Add 1×1 score layers for classification
4. Implement transposed convolutions for upsampling
5. Fuse features with element-wise addition

Expected Performance (PASCAL VOC)

- FCN-32s: 59% mIoU
- FCN-16s: 61% mIoU
- FCN-8s: 63% mIoU
- Each skip adds 2% mIoU

Part 1: FCN Code Structure

```python
class FCN8s(nn.Module):
    """TODO: Implement FCN-8s with skip connections"""
    def __init__(self, n_classes=21):
        super().__init__()
        # Task 1.1: Load pretrained ResNet50
        resnet = models.resnet50(pretrained=True)

        # Extract encoder layers
        self.conv1 = resnet.conv1
        self.bn1 = resnet.bn1
        self.relu = resnet.relu
        self.maxpool = resnet.maxpool
        self.layer1 = resnet.layer1  # 1/4
        self.layer2 = resnet.layer2  # 1/8 - pool3 for skip
        self.layer3 = resnet.layer3  # 1/16 - pool4 for skip
        self.layer4 = resnet.layer4  # 1/32

        # Task 1.2: Add score layers (1x1 conv to n_classes)
        self.score_pool3 = None  # TODO: Conv2d(1024, n_classes, 1)
        self.score_pool4 = None  # TODO: Conv2d(2048, n_classes, 1)
        self.score_fr = None     # TODO: Conv2d(2048, n_classes, 1)

        # Task 1.3: Add upsampling layers
        self.upscore2 = None        # TODO: ConvTranspose2d
        self.upscore_pool4 = None   # TODO: ConvTranspose2d
        self.upscore8 = None        # TODO: ConvTranspose2d

    def forward(self, x):
        # Task 1.4: Implement forward pass with progressive skip fusion
        pass
```

Part 1: FCN-8s Forward Pass — Pseudocode + Intuition

Algorithm 1: FCN-8s Forward Pass with Skip Connections

Data: Input image $\mathbf{I} \in \mathbb{R}^{B\times 3\times H\times W}$
Result: Segmentation map $\mathbf{S} \in \mathbb{R}^{B\times K\times H\times W}$
Encoder Path:
$\mathbf{x} \leftarrow \mathrm{relu}(\mathrm{bn1}(\mathrm{conv1}(\mathbf{I})))$;
$\mathbf{x} \leftarrow \mathrm{maxpool}(\mathbf{x})$;
$\mathbf{x} \leftarrow \mathrm{layer1}(\mathbf{x})$ `// stride 4`
pool3 $\leftarrow \mathrm{layer2}(\mathbf{x})$ `// stride 8, save for skip`
pool4 $\leftarrow \mathrm{layer3}(\mathbf{pool3})$ `// stride 16, save for skip`
$\mathbf{x} \leftarrow \mathrm{layer4}(\mathbf{pool4})$ `// stride 32`

Score Layers (1×1 convolutions):
score_fr $\leftarrow \mathrm{Conv}_{1\times 1}(\mathbf{x}, 2048 \to K)$;
score_pool4 $\leftarrow \mathrm{Conv}_{1\times 1}(\mathbf{pool4}, 1024/2048 \to K)$;
score_pool3 $\leftarrow \mathrm{Conv}_{1\times 1}(\mathbf{pool3}, 512/1024 \to K)$;

Progressive Upsampling with Skips:
`// First skip: pool4 at stride 16`
upscore2 $\leftarrow \mathrm{ConvTranspose2d}(\mathbf{score\_fr}, \mathrm{stride}=2)$ `// 32 → 16`
fuse_pool4 $\leftarrow$ **upscore2** + **score_pool4** `// elementwise add`

`// Second skip: pool3 at stride 8`
upscore_pool4 $\leftarrow \mathrm{ConvTranspose2d}(\mathbf{fuse\_pool4}, \mathrm{stride}=2)$ `// 16 → 8`
fuse_pool3 $\leftarrow$ **upscore_pool4** + **score_pool3** `// elementwise add`

`// Final upsampling to original resolution`
$\mathbf{S} \leftarrow \mathrm{ConvTranspose2d}(\mathbf{fuse\_pool3}, \mathrm{stride}=8)$ `// 8 → 1 (H,W)`

return S;

PyTorch Implementation Notes

- **1×1 score layers**: `nn.Conv2d(C_in, K, kernel_size=1)` maps feature channels → classes.
- **Upsampling**: Use `nn.ConvTranspose2d(K,K,4,2,1)` for ×2; for ×8: `kernel_size=16,stride=8,padding=4`.
- **Shape alignment**: Ensure skip tensors have same `H×W` and channels $= K$ before addition.
- **Alt. upsampling**: `F.interpolate(..., scale_factor=2, mode='bilinear', align_corners=False)` + 1×1 convs is OK (simpler, fewer params).
- **Padding/stride**: Mismatched sizes? Use `center_crop` or `F.pad` before addition.
- **Normalization**: Keep the ResNet stem (conv1/bn1/relu/maxpool) as-is for correct strides.
- **Initialization**: Kaiming for new convs; optionally freeze early ResNet layers then unfreeze.

Typical Tensor Shapes (example H=W=512)

$$\mathbf{I} : B \times 3 \times 512 \times 512$$
$$\mathrm{layer1} : B \times 256 \times 128 \times 128\ (/4)$$
$$\mathbf{pool3} = \mathrm{layer2} : B \times 512 \times 64 \times 64\ (/8)$$
$$\mathbf{pool4} = \mathrm{layer3} : B \times 1024 \times 32 \times 32\ (/16)$$
$$\mathrm{layer4} : B \times 2048 \times 16 \times 16\ (/32)$$
$$\mathbf{S} : B \times K \times 512 \times 512$$

Common Pitfalls

- Adding tensors with different `H×W` (off-by-one from padding).
- Forgetting to set `bias=False` on 1×1 score layers if followed by BN (optional).
- Training only with `CrossEntropy` → slow boundary recovery; try **CE + Dice**.
- Class imbalance in VOC: consider `ignore_index=255` and class weights.

Part 2: DeepLabV3+ with ASPP

Multi-Scale Context via Atrous Convolutions

ASPP (Atrous Spatial Pyramid Pooling) captures multi-scale context without losing resolution by using parallel atrous convolutions at different dilation rates.

Atrous Convolution

Standard conv:

$$y[i,j] = \sum_{m,n} x[i+m, j+n] \cdot w[m,n]$$

Atrous conv (rate r):

$$y[i,j] = \sum_{m,n} x[i+r\cdot m, j+r\cdot n] \cdot w[m,n]$$

Effective kernel size:

$$k_{\text{eff}} = k + (k-1)(r-1)$$

For 3×3 kernel: $r = 6 \to 13 \times 13$ effective

ASPP Module (5 branches)

1. 1×1 convolution (point-wise features)
2. 3×3 atrous conv, rate=6
3. 3×3 atrous conv, rate=12
4. 3×3 atrous conv, rate=18
5. Global average pooling + 1×1 conv

All outputs concatenated → 1×1 projection

DeepLabV3+ Decoder

- 4× bilinear upsample from ASPP
- Concatenate with low-level features (stride 4)
- 3×3 conv refinement
- Final 4× upsampling to original size

Part 2: ASPP Module Code Structure

```python
class ASPP(nn.Module):
    """TODO: Implement Atrous Spatial Pyramid Pooling"""
    def __init__(self, in_channels, out_channels=256, rates=[6, 12, 18]):
        super().__init__()

        # Task 2.1: Branch 1 - 1x1 convolution
        self.conv1 = None  # TODO: Sequential(Conv2d, BatchNorm, ReLU)

        # Task 2.1: Branches 2-4 - Atrous convolutions
        self.atrous_convs = nn.ModuleList()
        # TODO: for rate in rates: append Conv2d with dilation=rate

        # Task 2.1: Branch 5 - Global pooling
        self.global_avg_pool = None  # TODO: AdaptiveAvgPool + Conv

        # Task 2.2: Fusion layer (5 * out_channels -> out_channels)
        self.conv_out = None  # TODO: Conv + BN + ReLU + Dropout

    def forward(self, x):
        # Task 2.2: Apply all branches and concatenate
        pass
```

Part 2: ASPP — Pseudocode + Intuition

Algorithm 2: ASPP Module Implementation

Data: Feature map $\mathbf{X} \in \mathbb{R}^{B\times C\times H\times W}$, Rates $\{6, 12, 18\}$
Result: Multi-scale features $\mathbf{Y} \in \mathbb{R}^{B\times 256\times H\times W}$
size $\leftarrow (H, W)$ `// Save spatial dimensions`

Branch 1: 1×1 Convolution
$\mathbf{f}_0 \leftarrow \mathrm{Conv}_{1\times1}(\mathbf{X}, C \to 256)$;
$\mathbf{f}_0 \leftarrow \mathrm{BN}(\mathbf{f}_0) \to \mathrm{ReLU}(\mathbf{f}_0)$;

Branches 2–4: Atrous Convolutions
for $i = 1$ *to* 3 **do**
  $r_i \leftarrow \{6, 12, 18\}[i]$ `// Dilation rate`
  $\mathbf{f}_i \leftarrow \mathrm{Conv}_{3\times3}(\mathbf{X},$ dilation $= r_i,$ padding $= r_i)$;
  $\mathbf{f}_i \leftarrow \mathrm{BN}(\mathbf{f}_i) \to \mathrm{ReLU}(\mathbf{f}_i)$;
end

Branch 5: Global Average Pooling
$\mathbf{f}_g \leftarrow \mathrm{AdaptiveAvgPool2d}(\mathbf{X}, 1)$;
$\mathbf{f}_g \leftarrow \mathrm{Conv}_{1\times1}(\mathbf{f}_g, C \to 256)$;
$\mathbf{f}_g \leftarrow \mathrm{BN}(\mathbf{f}_g) \to \mathrm{ReLU}(\mathbf{f}_g)$;
$\mathbf{f}_4 \leftarrow \mathrm{Interpolate}(\mathbf{f}_g,$ size $= (H, W))$;

Feature Fusion
$\mathbf{Y}_{cat} \leftarrow \mathrm{Concat}([\mathbf{f}_0, \mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3, \mathbf{f}_4], \mathrm{dim} = 1)$;
$\mathbf{Y} \leftarrow \mathrm{Conv}_{1\times1}(\mathbf{Y}_{cat}, 1280 \to 256)$;
$\mathbf{Y} \leftarrow \mathrm{BN}(\mathbf{Y}) \to \mathrm{ReLU}(\mathbf{Y}) \to \mathrm{Dropout}(\mathbf{Y}, p = 0.1)$;
return $\mathbf{Y}$;

PyTorch Implementation Notes

- **Dilated 3×3**: set `padding=dilation` to keep $H \times W$.
- **Rates vs output stride**: with *OS=16*, typical rates are {6,12,18}; with *OS=8*, halve them (e.g., {3,6,9}) to avoid gridding.
- **Branch 5 upsampling**: `F.interpolate(fg, size=(H,W), mode='bilinear', align_corners=False)`.
- **Concat dim**: channel axis = `dim=1` → $5 \times 256 = 1280$ before the 1×1 projection.
- **Separable conv (optional)**: DeepLabV3+ often uses depthwise-separable 3×3 for speed/memory.
- **Norm choice**: small batch? prefer SyncBN/GroupNorm over BN to stabilize training.
- **Dropout**: light regularization (`p=0.1`) after the projection is standard.

Typical Shapes (ResNet, OS=16)

$$\mathbf{X} : B \times C \times 32 \times 32 \quad (H{=}W{=}32 \text{ if input } 512)$$
$$\mathbf{f}_0 \ldots \mathbf{f}_4 : B \times 256 \times 32 \times 32$$
$$\mathbf{Y}_{cat} : B \times 1280 \times 32 \times 32$$
$$\mathbf{Y} : B \times 256 \times 32 \times 32$$

Common Pitfalls

- **Wrong padding** with dilation ⇒ spatial mismatch on concat.
- **Too-large rates** at small H, W ⇒ aliasing/gridding; adjust by OS.
- `align_corners=True` can distort scaling; keep `False` unless you know why.
- Forgetting to **project low-level features** (e.g., 48 ch) in the decoder before concat.

Part 3: Mini-SAM - Trainable from Scratch

Educational Prompt-Based Segmentation

Mini-SAM is a simplified, trainable version of SAM with 5M parameters (vs 636M) that can be trained from scratch on limited data while demonstrating the key concepts of prompt-based segmentation.

Mini-SAM Architecture

Image Encoder (Lightweight):

- MobileNetV3-Small backbone
- Output: H/8 $\times$ W/8 $\times$ 256 feature map
- Fast encoding (50ms)

Prompt Encoder:

- Point encoding: type + position embeddings
- Box encoding: corner coordinates
- Broadcast to spatial dimensions

Mask Decoder (Conv-based):

- Fuse image + prompt features
- 3-layer CNN decoder
- Output: class logits + IoU prediction

Training Strategy

Simulated Prompts:

- Sample points from ground truth masks
- 50% foreground, 50% background points
- Random number of points (3-10)
- Optional box prompts from mask bounds

Loss Function:

$$\mathcal{L} = \mathcal{L}_{CE} + \mathcal{L}_{Dice} + 0.1 \cdot \mathcal{L}_{IoU}$$

Training Data:

- PASCAL VOC subset: 500 images
- Augmentation: flip, scale, crop
- Training time: 2-3 hours on single GPU

Key Advantages

Part 3: Mini-SAM Code Structure

```python
class MiniSAM(nn.Module):
    """TODO: Implement lightweight trainable SAM"""
    def __init__(self, n_classes=21, embed_dim=256):
        super().__init__()

        # Task 3.1: Lightweight Image Encoder (~2M params)
        backbone = models.mobilenet_v3_small(pretrained=True)
        self.image_encoder = None  # TODO: Extract features
        self.img_proj = None       # TODO: Project to embed_dim

        # Task 3.2: Simple Prompt Encoder
        self.point_type_embed = None  # TODO: Embedding(2, embed_dim)
        self.point_pos_embed = None   # TODO: MLP for position
        self.box_embed = None         # TODO: MLP for boxes

        # Task 3.3: Decoder
        self.decoder = None     # TODO: Conv layers for fusion
        self.mask_head = None   # TODO: Output n_classes
        self.iou_head = None    # TODO: IoU prediction

    def forward(self, images, points=None, point_labels=None, boxes=None):
        # Task 3.4: Implement forward pass
        # 1. Encode image
        # 2. Encode prompts
        # 3. Fuse features
        # 4. Decode mask + IoU
        pass
```

Part 3: Mini-SAM — Training Loop & Intuition

Algorithm 3: Mini-SAM Training with Simulated Prompts

Data: Images $\mathbf{I}$, Ground truth masks $\mathbf{M}_{gt}$
Result: Trained Mini-SAM model
Model Initialization:
Initialize MobileNetV3 encoder (pretrained);
Initialize prompt encoders (random);
Initialize decoder (random);

Training Loop:
for *each batch* $(\mathbf{I}_b, \mathbf{M}_{gt,b})$ **do**

```
// Simulate user prompts from ground truth
```
$\mathbf{P}, \mathbf{L} \leftarrow \text{SamplePoints}(\mathbf{M}_{gt,b}, n \in [3, 10], p_{fg} = 0.5)$;
$\mathbf{B} \leftarrow \text{SampleBoxesFromMasks}(\mathbf{M}_{gt,b})$ `// optional`

```
// Forward pass
```
$\mathbf{F}_{img} \leftarrow \text{ImageEncoder}(\mathbf{I}_b)$;
$\mathbf{F}_{prompt} \leftarrow \text{PromptEncoder}(\mathbf{P}, \mathbf{L}, \mathbf{B})$;
$\mathbf{F}_{fused} \leftarrow \text{Concat}(\mathbf{F}_{img}, \mathbf{F}_{prompt})$;
$(\mathbf{M}_{pred}, \widehat{\text{IoU}}) \leftarrow \text{Decoder}(\mathbf{F}_{fused})$;

```
// Compute losses
```
$\mathcal{L}_{CE} \leftarrow \text{CrossEntropy}(\mathbf{M}_{pred}, \mathbf{M}_{gt,b})$;
$\mathcal{L}_{Dice} \leftarrow 1 - \text{DiceCoeff}(\mathbf{M}_{pred}, \mathbf{M}_{gt,b})$;
$\text{IoU}_{true} \leftarrow \text{ComputeIoU}(\mathbf{M}_{pred}, \mathbf{M}_{gt,b})$;
$\mathcal{L}_{IoU} \leftarrow \text{MSE}(\widehat{\text{IoU}}, \text{stopgrad}(\text{IoU}_{true}))$;
$\mathcal{L}_{total} \leftarrow \mathcal{L}_{CE} + \mathcal{L}_{Dice} + 0.1 \cdot \mathcal{L}_{IoU}$;

```
// Backward pass and update
```
optimizer.zero_grad();
$\mathcal{L}_{total}$.backward();
clip_grad_norm(θ, τ) `// e.g.,` $\tau = 1.0$
optimizer.step();
scheduler.step$(\mathcal{L}_{val})$ `// optional`

end

return Trained model;

PyTorch Implementation Notes

- **Batching prompts**: fix N_{pts} per image (pad/truncate); tensors `points: (B,N,2)` in normalized coords $[0, 1]$ or pixel (x, y).
- **Point types**: `point_labels: (B,N)` with {0:bg, 1:fg}; embed via `nn.Embedding(2, D)`.
- **Box prompts**: `boxes: (B,4)` as $(x_{min}, y_{min}, x_{max}, y_{max})$, normalize to $[0, 1]$, project with MLP to D.
- **Broadcasting**: tile prompt embeddings to `(B,D,H/8,W/8)` and concat with image features along `dim=1`.
- **Shapes (512×512, OS=8)**:

$$\mathbf{F}_{img} : B \times 256 \times 64 \times 64, \quad \mathbf{F}_{prompt} : B \times 256 \times 64 \times 64, \quad \mathbf{F}_{fused} : B \times 512 \times 64 \times 64.$$

- **Losses**: `CrossEntropyLoss(ignore_index=255)` on *logits* (no softmax first). Dice on softmax probs.
- **IoU head**: predicts scalar `(B,1)` or per-class `(B,K)`; compute *target IoU* with `no_grad`.

Stability, Speed, and Tricks

- **Warmup**: freeze backbone for 1–3 epochs, then unfreeze.
- **Optim**: AdamW (`lr=3e-4`, `wd=1e-4`); cosine or ReduceLROnPlateau.
- **AMP**: use `torch.cuda.amp.autocast` + `GradScaler` for 1.5–2× speed.
- **Grad clip**: `clip_grad_norm_` to avoid exploding grads in decoder.
- **Augment**: random flip/scale/crop; keep points/boxes consistent with transforms.

Common Pitfalls

- Mismatch between transformed images and untransformed prompts ⇒ wrong supervision.
- Applying softmax before `CrossEntropyLoss` (double-softmax bug).
- Computing IoU with logits instead of thresholded/argmax masks for the target signal.
- Forgetting to normalize point/box coordinates to the encoder stride (OS).

Loss Functions and Evaluation Metrics

Loss Function Implementations

Cross-Entropy: $\mathcal{L}_{CE} = -\frac{1}{N}\sum_{i,c} y_{i,c}\log(\hat{y}_{i,c})$
Dice Loss: $\mathcal{L}_{Dice} = 1 - \frac{2\sum y\hat{y}+\epsilon}{\sum y+\sum \hat{y}+\epsilon}$
Focal Loss: $\mathcal{L}_{Focal} = -\alpha(1-p_t)^{\gamma}\log(p_t)$ where $p_t = \begin{cases} p & \text{if } y=1 \\ 1-p & \text{else} \end{cases}$
IoU Prediction Loss: $\mathcal{L}_{IoU} = \mathrm{MSE}(\mathrm{IoU}_{pred}, \mathrm{IoU}_{true})$
Combined: $\mathcal{L} = \lambda_1\mathcal{L}_{CE} + \lambda_2\mathcal{L}_{Dice} + \lambda_3\mathcal{L}_{IoU}$

Evaluation Metrics

Mean IoU: $\mathrm{mIoU} = \frac{1}{K}\sum_{k=1}^{K}\frac{|A_k\cap B_k|}{|A_k\cup B_k|}$
Pixel Accuracy: $\mathrm{PA} = \frac{\sum_k n_{kk}}{\sum_k t_k}$ where n_{kk} = correctly classified pixels
IoU Accuracy: Correlation between predicted and true IoU scores

Training Configuration and Best Practices

Hyperparameters

FCN/DeepLab:

- AdamW, LR: 1e-4, weight decay 1e-4
- Batch size: 8-16
- Epochs: 50

Mini-SAM:

- AdamW, LR: 1e-4
- Batch size: 16
- Epochs: 30
- Points per sample: 3-10 (random)

Learning Rate Schedule:

- ReduceLROnPlateau with patience=5
- Or polynomial decay

Data Augmentation

- Random crop to 256×256
- Random horizontal flip (p=0.5)
- Random scale (0.8 to 1.2)
- Color jitter
- Normalize: ImageNet statistics

Validation Strategy

- Evaluate every epoch
- Save best mIoU checkpoint
- Track per-class IoU
- Visualize predictions
- For Mini-SAM: test with different prompt counts

Debugging Tips

- Start with 1 image overfitting
- Check tensor shapes
- Visualize prompt sampling
- Use synthetic data first

Expected Results and Performance Comparison

Quantitative Results (PASCAL VOC 2012 val / subset)

Model	mIoU (%)	Params (M)	Inference (ms)	Training Time
FCN-32s	59.4	140	35	2-3 hours
FCN-16s	61.2	140	38	2-3 hours
FCN-8s	63.1	140	42	3-4 hours
DeepLabV3+	77.5	60	80	6-8 hours
Mini-SAM (ours)	**65-68**	**5**	**60**	**2-3 hours**
For reference (not implemented in lab):				
SAM (zero-shot)	75.0	636	1050	N/A
SAM (fine-tuned)	82.0	636	1050	2-3 hours

Key Observations

- **FCN**: Skip connections provide consistent improvement (+3.7% from FCN-32s to FCN-8s)
- **DeepLabV3+**: ASPP multi-scale context gives major boost (+14.4% over FCN-8s)
- **Mini-SAM**: Competitive with FCN-8s despite being trained from scratch with limited data!
- **Efficiency**: Mini-SAM has 28× fewer parameters than FCN, 127× fewer than full SAM
- **Pedagogical value**: Only Mini-SAM shows complete training pipeline

Comparative Analysis and Use Cases

When to Use Each Method

FCN-8s:

- Real-time applications (40ms inference)
- Good baseline for research
- Simple architecture for learning

DeepLabV3+:

- Production segmentation systems
- Best performance/speed trade-off
- Moderate computational budget

Mini-SAM:

- Understanding prompt-based segmentation
- Limited training data scenarios
- Interactive applications (point/box prompts)
- Rapid prototyping and experimentation
- Educational demonstrations

Mini-SAM vs Full SAM

Advantages of Mini-SAM for learning:

- Can be trained from scratch in lab timeframe
- Understandable architecture (5M params)
- Shows complete workflow: data $\rightarrow$ training $\rightarrow$ inference
- Demonstrates key concepts without requiring massive compute

Deliverables and Grading Rubric

Required Implementations (100 points)

1. **FCN Implementation (25 points):**
   - Working FCN-32s, FCN-16s, FCN-8s (12 points)
   - Training curves showing improvement (5 points)
   - Qualitative visualization (4 points)
   - Ablation study (4 points)
2. **DeepLabV3+ Implementation (25 points):**
   - Complete ASPP module (12 points)
   - Encoder-decoder architecture (8 points)
   - Performance comparison with FCN (5 points)
3. **Mini-SAM Implementation (30 points):**
   - Architecture implementation (10 points)
   - Prompt sampling and encoding (8 points)
   - Training from scratch (8 points)
   - Interactive demo with point/box prompts (4 points)
4. **Analysis and Report (20 points):**
   - Quantitative comparison table (8 points)
   - Discussion of trade-offs (6 points)
   - Code quality and documentation (6 points)

Submission Guidelines

What to Submit

1. **Code Files:**
   - `lab05_student.py` - All implementations
   - `train_fcn.py` - FCN training script (optional separate file)
   - `train_deeplab.py` - DeepLab training script (optional)
   - `train_minisam.py` - Mini-SAM training script (optional)
   - `demo_minisam.ipynb` - Interactive demo notebook
2. **Model Checkpoints:**
   - Best FCN-8s model (.pth)
   - Best DeepLabV3+ model (.pth)
   - Best Mini-SAM model (.pth)
   - Training logs (TensorBoard or CSV)
3. **Report (PDF):**
   - 3-5 pages including figures and tables
   - Architecture descriptions
   - Quantitative results table (all 3 methods)
   - Qualitative predictions visualization
   - Discussion: Why does Mini-SAM work despite limited training?

Deadline and Format

Due: Two weeks from today, 23:59
Format: Single ZIP file uploaded to course platform
Naming: `lab05_[lastname]_[firstname].zip`

Common Pitfalls and Solutions

FCN Issues

Dimension mismatch in skip addition:

- Check spatial sizes match exactly
- Use center cropping if needed

Checkerboard artifacts:

- Initialize transposed conv with bilinear
- Try bilinear upsample + conv instead

DeepLabV3+ Issues

ASPP checkerboard:

- Ensure padding = dilation rate
- Check all branches have BN

Out of memory:

- Reduce batch size to 4
- Lower ASPP channels to 128

Mini-SAM Issues

Poor convergence:

- Check prompt sampling (balanced fg/bg)
- Verify normalization of point coordinates
- Start with more points (8-10)
- Use pretrained MobileNet encoder

IoU prediction always wrong:

- Ensure IoU computed correctly
- Check loss weight (should be 0.1)
- Normalize IoU to [0, 1]

Features not aligning:

- Verify prompt broadcasting to spatial size
- Check concatenation dimensions
- Print intermediate shapes

Debugging strategy:

- Test with 1 class first
- Visualize sampled prompts
- Use synthetic circles/squares

Resources and References

Key Papers

- **FCN**: Long et al., "Fully Convolutional Networks for Semantic Segmentation," CVPR 2015
- **DeepLabV3+**: Chen et al., "Encoder-Decoder with Atrous Separable Convolution," ECCV 2018
- **SAM**: Kirillov et al., "Segment Anything," ICCV 2023
- **MobileNetV3**: Howard et al., "Searching for MobileNetV3," ICCV 2019

Code and Datasets

Official Implementations:

- SAM: `https://github.com/facebookresearch/segment-anything`
- torchvision FCN: `https://pytorch.org/vision/stable/models.html`
- Our Mini-SAM: See artifact code provided earlier

Datasets:

- PASCAL VOC 2012: `http://host.robots.ox.ac.uk/pascal/VOC/voc2012/`
- Create subset: `random.sample(train_data, 500)`

Help

- Office hours: by appointment
- Course forum for questions