Artificial Vision (VIAR25/26)

Lab 4: Encoder-Decoders: Building and Analyzing U-Net for Dense Prediction

Prof. David Olivieri

UVigo

September 29, 2025

Lab Objectives

Today's Mission

Implement, analyze, and optimize encoder-decoder architectures for semantic segmentation. You'll build a modular U-Net, experiment with skip connections, and solve a real segmentation task.

Learning Outcomes

1. **Build** a complete U-Net architecture from scratch
2. **Implement** different skip connection strategies and compare performance
3. **Experiment** with various loss functions for dense prediction
4. **Optimize** memory usage and inference speed
5. **Evaluate** using proper segmentation metrics

Dataset

Oxford-IIIT Pet Dataset - Semantic segmentation

- 37 pet categories with pixel-level masks
- Binary segmentation: pet vs. background
- Images: 200x200 pixels (resized for efficiency)

Lab Structure

Part 1: Core U-Net Implementation (40 min)

- **Task 1.1**: Implement encoder blocks with proper downsampling
- **Task 1.2**: Implement decoder blocks with upsampling
- **Task 1.3**: Connect encoder-decoder with skip connections
- **Task 1.4**: Create configurable U-Net class

Part 2: Skip Connection Experiments (30 min)

- **Task 2.1**: Implement concatenation-based skip connections
- **Task 2.2**: Implement addition-based skip connections
- **Task 2.3**: Implement attention-gated skip connections
- **Task 2.4**: Compare gradient flow and performance

Part 3: Loss Functions and Training (30 min)

- **Task 3.1**: Implement Dice loss
- **Task 3.2**: Implement focal loss for class imbalance
- **Task 3.3**: Create combined loss function
- **Task 3.4**: Train and evaluate models

Part 1: Building U-Net Architecture

Core Components to Implement

You'll build a modular U-Net where each component can be tested independently.

Encoder Block Requirements

- Double convolution with BatchNorm
- ReLU activation
- MaxPooling for downsampling
- Store features for skip connections
- Channel doubling: 64→128→256→512

Decoder Block Requirements

- Upsampling (transpose conv or bilinear)
- Skip connection fusion
- Double convolution
- Channel reduction: 512→256→128→64
- Handle dimension mismatches

Implementation Hints

- Use `nn.Sequential` for conv blocks
- Return both pooled and unpooled features
- Consider padding to maintain dimensions

Key Challenges

- Matching tensor dimensions for concatenation
- Choosing upsampling method
- Memory efficiency with large feature maps

Part 1: Code Structure

```python
class EncoderBlock(nn.Module):
    """TODO: Implement double conv + pooling"""
    def __init__(self, in_channels, out_channels):
        super().__init__()
        # Task 1.1: Define your layers here

    def forward(self, x):
        # Task 1.1: Implement forward pass
        # Return: features before pooling (for skip), features after pooling
        pass

class DecoderBlock(nn.Module):
    """TODO: Implement upsampling + skip connection + double conv"""
    def __init__(self, in_channels, skip_channels, out_channels):
        super().__init__()
        # Task 1.2: Define your layers here

    def forward(self, x, skip_features):
        # Task 1.2: Implement forward pass with skip connection
        pass

class UNet(nn.Module):
    """TODO: Complete U-Net architecture"""
    def __init__(self, in_channels=3, out_channels=1, features=[64, 128, 256, 512]):
        super().__init__()
        # Task 1.3: Build encoder and decoder paths

    def forward(self, x):
        # Task 1.4: Connect everything together
        pass

```

Part 1.1: Double Convolution Block Algorithm

Core Building Block

The double convolution block is the fundamental unit of U-Net, appearing in both encoder and decoder paths.

Algorithm 1: DoubleConv: Two convolutions with BatchNorm and ReLU

Data: Input tensor $\mathbf{X} \in \mathbb{R}^{B \times C_{in} \times H \times W}$
Result: Output tensor $\mathbf{Y} \in \mathbb{R}^{B \times C_{out} \times H \times W}$

Parameters to Initialize:
$\mathbf{W}_1 \in \mathbb{R}^{C_{out} \times C_{in} \times 3 \times 3}$ `// First conv weights`
$\mathbf{b}_1 \in \mathbb{R}^{C_{out}}$ `// First conv bias`
$\mathbf{W}_2 \in \mathbb{R}^{C_{out} \times C_{out} \times 3 \times 3}$ `// Second conv weights`
$\mathbf{b}_2 \in \mathbb{R}^{C_{out}}$ `// Second conv bias`
$\gamma_1, \beta_1, \gamma_2, \beta_2$ `// BatchNorm parameters`

Forward Pass:
$\mathbf{H}_1 \leftarrow \texttt{Conv2D}(\mathbf{X}, \mathbf{W}_1, \mathbf{b}_1, \text{padding} = 1)$;
$\mathbf{H}_2 \leftarrow \texttt{BatchNorm2D}(\mathbf{H}_1, \gamma_1, \beta_1)$;
$\mathbf{H}_3 \leftarrow \texttt{ReLU}(\mathbf{H}_2)$;
$\mathbf{H}_4 \leftarrow \texttt{Conv2D}(\mathbf{H}_3, \mathbf{W}_2, \mathbf{b}_2, \text{padding} = 1)$;
$\mathbf{H}_5 \leftarrow \texttt{BatchNorm2D}(\mathbf{H}_4, \gamma_2, \beta_2)$;
$\mathbf{Y} \leftarrow \texttt{ReLU}(\mathbf{H}_5)$;
return $\mathbf{Y}$;

Implementation Notes

- Padding=1 maintains spatial dimensions: $H_{out} = H_{in}$, $W_{out} = W_{in}$
- BatchNorm normalizes across the batch dimension for each channel
- Use `nn.Sequential` in PyTorch for cleaner implementation

Part 1.2: Encoder Block Algorithm

Algorithm 2: EncoderBlock: Feature extraction with downsampling

Data: Input tensor $\mathbf{X} \in \mathbb{R}^{B \times C_{in} \times H \times W}$

Result: Skip features $\mathbf{F}_{skip}$, Pooled features $\mathbf{F}_{pool}$

Encoder Block Processing:

`// Apply double convolution to extract features`

$\mathbf{F}_{skip} \leftarrow \texttt{DoubleConv}(\mathbf{X}, C_{in} \rightarrow C_{out})$;

`// Store` $\mathbf{F}_{skip}$ `for later skip connection`

`// Downsample spatially by factor of 2`

$\mathbf{F}_{pool} \leftarrow \texttt{MaxPool2D}(\mathbf{F}_{skip}, \text{kernel} = 2, \text{stride} = 2)$;

`// Output dimensions:`

`//` $\mathbf{F}_{skip} \in \mathbb{R}^{B \times C_{out} \times H \times W}$

`//` $\mathbf{F}_{pool} \in \mathbb{R}^{B \times C_{out} \times H/2 \times W/2}$

return $\mathbf{F}_{skip}, \mathbf{F}_{pool}$;

Encoder Path Structure

- Each level doubles channels
- Each level halves spatial dims
- Skip connections preserve details

Channel Progression

- Level 1: $3 \rightarrow 64$
- Level 2: $64 \rightarrow 128$
- Level 3: $128 \rightarrow 256$
- Level 4: $256 \rightarrow 512$

Part 1.3: Decoder Block Algorithm

Algorithm 3: DecoderBlock: Upsampling with skip connection fusion

Data: Decoder input $\mathbf{X} \in \mathbb{R}^{B \times C_{in} \times H \times W}$, Skip features $\mathbf{F}_{skip} \in \mathbb{R}^{B \times C_{skip} \times 2H \times 2W}$
Result: Decoded features $\mathbf{Y} \in \mathbb{R}^{B \times C_{out} \times 2H \times 2W}$

Upsampling Step:
if *mode* $=$ *"transpose"* **then**
  $\mathbf{X}_{up} \leftarrow \texttt{ConvTranspose2D}(\mathbf{X}, C_{in} \rightarrow C_{in}/2, \text{kernel} = 2, \text{stride} = 2)$;
else
  $\mathbf{X}_{up} \leftarrow \texttt{Upsample}(\mathbf{X}, \text{scale} = 2, \text{mode} = \text{bilinear})$;
  $\mathbf{X}_{up} \leftarrow \texttt{Conv2D}(\mathbf{X}_{up}, C_{in} \rightarrow C_{in}/2, \text{kernel} = 1)$;
end

Handle Dimension Mismatch:
if $\mathbf{X}_{up}.shape[2:] \neq \mathbf{F}_{skip}.shape[2:]$ **then**
  $\mathbf{X}_{up} \leftarrow \texttt{CropOrPad}(\mathbf{X}_{up}, \text{target\_shape} = \mathbf{F}_{skip}.\text{shape})$;
end

Skip Connection Fusion:
$\mathbf{X}_{concat} \leftarrow \texttt{Concatenate}([\mathbf{X}_{up}, \mathbf{F}_{skip}], \text{dim} = 1)$;
// Combined channels: $C_{in}/2 + C_{skip}$

Feature Refinement:
$\mathbf{Y} \leftarrow \texttt{DoubleConv}(\mathbf{X}_{concat}, (C_{in}/2 + C_{skip}) \rightarrow C_{out})$;

Part 1.4: Complete U-Net Forward Pass

Algorithm 4: U-Net: Complete encoder-decoder architecture

Data: Input image $\mathbf{I} \in \mathbb{R}^{B \times 3 \times H \times W}$
Result: Segmentation map $\mathbf{S} \in \mathbb{R}^{B \times N_{classes} \times H \times W}$
Initialize: skip_connections $\leftarrow$ [];
features $= [64, 128, 256, 512]$;

Encoder Path:
$\mathbf{x} \leftarrow \mathbf{I}$;
for $i = 0$ *to* $len(features) - 1$ **do**
  $C_{in} \leftarrow 3$ if $i = 0$ else features$[i-1]$;
  $C_{out} \leftarrow$ features$[i]$;
  $\mathbf{skip}_i, \mathbf{x} \leftarrow$ EncoderBlock$(\mathbf{x}, C_{in}, C_{out})$;
  skip_connections.append$(\mathbf{skip}_i)$;
end

Bottleneck:
$\mathbf{x} \leftarrow$ DoubleConv$(\mathbf{x}, 512 \rightarrow 1024)$;

Decoder Path:
for $i = len(features) - 1$ *down to* 0 **do**
  $C_{in} \leftarrow 1024$ if $i =$ len(features) $- 1$ else features$[i+1]$;
  $C_{out} \leftarrow$ features$[i]$;
  $\mathbf{x} \leftarrow$ DecoderBlock$(\mathbf{x}$, skip_connections$[i], C_{in}, C_{out})$;
end

Part 2: Skip Connection Strategies

Comparing Information Flow Mechanisms

Different skip connection strategies affect gradient flow, memory usage, and final performance.

Three Strategies to Implement

Concatenation

- Direct feature stacking
- Preserves all information
- Higher memory usage
- $[\mathbf{F}_{\text{up}}, \mathbf{F}_{\text{skip}}]$

Addition

- Element-wise sum
- Memory efficient
- May lose information
- $\mathbf{F}_{\text{up}} + \mathbf{F}_{\text{skip}}$

Attention-Gated

- Learned importance
- Focus on relevant features
- Computational overhead
- $\alpha \odot \mathbf{F}_{\text{skip}}$

Analysis Requirements

- Measure gradient magnitude at different depths
- Compare memory consumption
- Evaluate final segmentation IoU
- Visualize attention maps (for attention-gated)

Part 2.1: Concatenation Skip Connection

Algorithm 5: Concatenation-based skip connection

Data: Decoder features $\mathbf{D} \in \mathbb{R}^{B \times C_d \times H \times W}$, Skip features $\mathbf{S} \in \mathbb{R}^{B \times C_s \times H \times W}$
Result: Fused features $\mathbf{F} \in \mathbb{R}^{B \times C_{out} \times H \times W}$

Concatenation Fusion:

// Concatenate along channel dimension
$\mathbf{F}_{concat} \leftarrow \texttt{Concatenate}([\mathbf{D}, \mathbf{S}], \text{dim} = 1)$;
// Output shape: $B \times (C_d + C_s) \times H \times W$

// Reduce channels to desired output size
$\mathbf{F} \leftarrow \texttt{Conv2D}(\mathbf{F}_{concat}, (C_d + C_s) \rightarrow C_{out}, \text{kernel} = 3, \text{padding} = 1)$;

return F;

Properties

Advantages:

- Preserves all information
- No information loss
- Standard U-Net approach

Disadvantages:

- Higher memory usage
- More parameters in fusion conv
- Increased computation

Part 2.2: Addition Skip Connection

Algorithm 6: Addition-based skip connection (ResNet-style)

Data: Decoder features $\mathbf{D} \in \mathbb{R}^{B \times C_d \times H \times W}$, Skip features $\mathbf{S} \in \mathbb{R}^{B \times C_s \times H \times W}$
Result: Fused features $\mathbf{F} \in \mathbb{R}^{B \times C_d \times H \times W}$

Addition Fusion:

```
// Match channel dimensions if necessary
```
if $C_s \neq C_d$ **then**
  $\mathbf{S}_{proj} \leftarrow \text{Conv2D}(\mathbf{S}, C_s \rightarrow C_d, \text{kernel} = 1)$;
  `// 1×1 convolution for channel projection`
else
  $\mathbf{S}_{proj} \leftarrow \mathbf{S}$;
end

`// Element-wise addition`
$\mathbf{F} \leftarrow \mathbf{D} + \mathbf{S}_{proj}$;

return $\mathbf{F}$;

Properties

Advantages:

- Memory efficient
- Gradient flow like ResNet
- Fewer parameters

Disadvantages:

- Potential information loss
- Requires channel matching
- May lose fine details

Part 2.3: Attention-Gated Skip Connection

Algorithm 7: Attention-gated skip connection

Data: Gate signal $\mathbf{G} \in \mathbb{R}^{B \times C_g \times H_g \times W_g}$, Skip features $\mathbf{S} \in \mathbb{R}^{B \times C_s \times H \times W}$
Result: Attended features $\mathbf{A} \in \mathbb{R}^{B \times C_s \times H \times W}$

Attention Mechanism:

$C_{inter} \leftarrow C_g/2$ // Intermediate channels

// Transform gate signal
$\mathbf{W}_g \leftarrow \texttt{Conv2D}(\mathbf{G}, C_g \rightarrow C_{inter}, \text{kernel} = 1)$;

// Transform skip features
$\mathbf{W}_s \leftarrow \texttt{Conv2D}(\mathbf{S}, C_s \rightarrow C_{inter}, \text{kernel} = 1)$;

// Align spatial dimensions if needed
if $H_g \neq H$ *or* $W_g \neq W$ **then**
  $\mathbf{W}_g \leftarrow \texttt{Interpolate}(\mathbf{W}_g, \text{size} = (H, W))$;
end

// Compute attention coefficients
$\mathbf{Q} \leftarrow \texttt{ReLU}(\mathbf{W}_g + \mathbf{W}_s)$;
$\boldsymbol{\alpha} \leftarrow \texttt{Sigmoid}(\texttt{Conv2D}(\mathbf{Q}, C_{inter} \rightarrow 1, \text{kernel} = 1))$;
// $\boldsymbol{\alpha} \in [0, 1]^{B \times 1 \times H \times W}$

// Apply attention to skip features
$\mathbf{A} \leftarrow \mathbf{S} \odot \boldsymbol{\alpha}$ // Element-wise multiplication

Part 3: Loss Functions for Segmentation

Dice Loss Implementation

Formula: $\mathcal{L}_{\text{Dice}} = 1 - \frac{2|Y \cap \hat{Y}| + \epsilon}{|Y| + |\hat{Y}| + \epsilon}$

Your Tasks:

- Handle multi-class case
- Add smoothing factor
- Consider class weighting

Focal Loss Implementation

Formula: $\mathcal{L}_{\text{Focal}} = -\alpha(1 - p_t)^{\gamma} \log(p_t)$

Your Tasks:

- Implement focusing parameter γ
- Add class balancing α
- Handle numerical stability

Combined Loss Strategy

Multi-task learning approach:

$$\mathcal{L} = \lambda_1 \mathcal{L}_{\text{CE}} + \lambda_2 \mathcal{L}_{\text{Dice}} + \lambda_3 \mathcal{L}_{\text{boundary}}$$

Implementation considerations:

- Dynamic weight scheduling
- Loss magnitude normalization
- Gradient scaling

Evaluation Metrics

- IoU (Intersection over Union)
- Dice coefficient
- Pixel accuracy
- Boundary F1 score

Training Pipeline

Data Augmentation Strategy

- Random horizontal flips
- Random rotations ($\pm 15°$)
- Elastic deformations
- Color jittering
- **Important**: Apply same transform to image and mask!

Training Configuration

Hyperparameters:

- Batch size: 16
- Learning rate: 0.001
- Optimizer: Adam with weight decay
- Scheduler: ReduceLROnPlateau
- Epochs: 50

Monitoring:

- Training/validation loss curves
- IoU progression
- Learning rate schedule
- Sample predictions every 5 epochs
- Memory usage tracking

Debugging Tips

- Start with small image size (64x64) for quick iteration
- Verify dimensions at each layer with dummy input
- Use gradient clipping to prevent explosion
- Visualize intermediate feature maps

Monitoring Training and Evaluation

What to Track During Training

- **Loss curves**: training vs. validation loss
- **Learning rate schedule**: how LR changes over epochs
- **Segmentation metrics**:
  - IoU (Intersection over Union)
  - Dice coefficient
  - Pixel accuracy
- **Model statistics**: number of parameters, memory usage
- **Qualitative results**: predicted masks every few epochs

Why is this Important?

Metrics and curves help diagnose **overfitting**, **underfitting**, and **optimization issues**.

Expected Outputs and Evaluation

Deliverables

1. **Complete U-Net implementation** with modular design
2. **Comparison table** of skip connection strategies:
   - Final IoU scores
   - Training time per epoch
   - Memory consumption
   - Gradient flow analysis
3. **Loss function analysis**:
   - Learning curves for each loss
   - Qualitative comparison of segmentation quality
4. **Ablation study** removing components:
   - No skip connections
   - No batch normalization
   - Different depths

Bonus Challenges

- Implement U-Net++ with dense skip connections
- Add deep supervision at multiple scales
- Implement efficient inference with sliding window
- Create interactive visualization of attention maps

Ablation Studies in Deep Learning

What is an Ablation Study?

A systematic way to evaluate the contribution of each component in a model by **removing, replacing, or modifying** it.

Why Do We Perform Ablations?

- Understand which parts of the architecture are essential
- Separate useful design choices from unnecessary complexity
- Provide **scientific justification** for model design
- Improve reproducibility and clarity in research

Examples for U-Net

- **No skip connections**: how important are they for fine details?
- **No batch normalization**: effect on stability and convergence
- **Depth variations**: shallow vs. deep U-Net
- **Alternative skip connections**: concat, add, attention

Part 3.1: Dice Loss Algorithm

Algorithm 8: Dice Loss for Segmentation

Data: Predictions $\hat{\mathbf{Y}} \in \mathbb{R}^{B\times C\times H\times W}$ (logits), Targets $\mathbf{Y} \in \{0,1\}^{B\times C\times H\times W}$, Smoothing factor $\epsilon = 10^{-6}$
Result: Dice loss $\mathcal{L}_{Dice} \in \mathbb{R}$

Binary Dice Loss (C=1):
`// Apply sigmoid to get probabilities`
$\mathbf{P} \leftarrow \text{Sigmoid}(\hat{\mathbf{Y}})$;

`// Flatten spatial dimensions`
$\mathbf{P}_{flat} \leftarrow \text{Flatten}(\mathbf{P})$ `// Shape:` $(B \times HW)$
$\mathbf{Y}_{flat} \leftarrow \text{Flatten}(\mathbf{Y})$ `// Shape:` $(B \times HW)$

`// Compute Dice coefficient`
intersection $\leftarrow \sum_{i,j} \mathbf{P}_{flat}[i,j] \cdot \mathbf{Y}_{flat}[i,j]$;
union $\leftarrow \sum_{i,j} \mathbf{P}_{flat}[i,j] + \sum_{i,j} \mathbf{Y}_{flat}[i,j]$;
dice $\leftarrow \frac{2\cdot\text{intersection}+\epsilon}{\text{union}+\epsilon}$;

$\mathcal{L}_{Dice} \leftarrow 1 - \text{dice}$;

Multi-class Dice Loss (C¿1):
$\mathcal{L}_{Dice} \leftarrow 0$;
for $c = 1$ *to* C **do**
  $\text{dice}_c \leftarrow \text{BinaryDice}(\hat{\mathbf{Y}}[:,c,:,:], \mathbf{Y}[:,c,:,:])$;
  $\mathcal{L}_{Dice} \leftarrow \mathcal{L}_{Dice} + \text{dice}_c$;
end
$\mathcal{L}_{Dice} \leftarrow \mathcal{L}_{Dice}/C$;
return $\mathcal{L}_{Dice}$;

Part 3.2: Focal Loss Algorithm

Algorithm 9: Focal Loss for Class Imbalance

Data: Predictions $\hat{\mathbf{Y}} \in \mathbb{R}^{B \times C \times H \times W}$, Targets $\mathbf{Y} \in \{0,1\}^{B \times C \times H \times W}$, Balancing factor $\alpha = 0.25$, Focusing parameter $\gamma = 2.0$
Result: Focal loss $\mathcal{L}_{Focal} \in \mathbb{R}$

Focal Loss Computation:

// Apply sigmoid for probabilities
$\mathbf{P} \leftarrow \text{Sigmoid}(\hat{\mathbf{Y}})$;

// Compute binary cross-entropy
$\text{BCE} \leftarrow -\mathbf{Y} \cdot \log(\mathbf{P} + \epsilon) - (1 - \mathbf{Y}) \cdot \log(1 - \mathbf{P} + \epsilon)$;

// Compute probability of correct class
$\mathbf{P}_t \leftarrow \mathbf{Y} \cdot \mathbf{P} + (1 - \mathbf{Y}) \cdot (1 - \mathbf{P})$;

// Apply focal term to focus on hard examples
$\text{focal\_term} \leftarrow (1 - \mathbf{P}_t)^{\gamma}$;

// Apply class balancing
$\alpha_t \leftarrow \mathbf{Y} \cdot \alpha + (1 - \mathbf{Y}) \cdot (1 - \alpha)$;

// Combine all terms
$\mathcal{L}_{Focal} \leftarrow \alpha_t \cdot \text{focal\_term} \cdot \text{BCE}$;

// Average over all pixels and batch
$\mathcal{L}_{Focal} \leftarrow \frac{1}{B \cdot H \cdot W} \sum_{b,h,w} \mathcal{L}_{Focal}[b, :, h, w]$;

return $\mathcal{L}_{Focal}$;

Part 3.3: Combined Loss Algorithm

Algorithm 10: Combined Loss Function

Data: Predictions $\hat{\mathbf{Y}}$, Targets $\mathbf{Y}$, Loss weights $\{w_{BCE}, w_{Dice}, w_{Focal}\}$
Result: Combined loss $\mathcal{L}_{total} \in \mathbb{R}$

Initialize:
$\mathcal{L}_{total} \leftarrow 0$;

Compute Individual Losses:
if $w_{BCE} > 0$ **then**
  $\mathcal{L}_{BCE} \leftarrow \text{BCEWithLogitsLoss}(\hat{\mathbf{Y}}, \mathbf{Y})$;
  $\mathcal{L}_{total} \leftarrow \mathcal{L}_{total} + w_{BCE} \cdot \mathcal{L}_{BCE}$;
end

if $w_{Dice} > 0$ **then**
  $\mathcal{L}_{Dice} \leftarrow \text{DiceLoss}(\hat{\mathbf{Y}}, \mathbf{Y})$;
  $\mathcal{L}_{total} \leftarrow \mathcal{L}_{total} + w_{Dice} \cdot \mathcal{L}_{Dice}$;
end

if $w_{Focal} > 0$ **then**
  $\mathcal{L}_{Focal} \leftarrow \text{FocalLoss}(\hat{\mathbf{Y}}, \mathbf{Y})$;
  $\mathcal{L}_{total} \leftarrow \mathcal{L}_{total} + w_{Focal} \cdot \mathcal{L}_{Focal}$;
end

return $\mathcal{L}_{total}$;

Weight Selection Guidelines

- Balanced dataset: $w_{BCE} = 0.5, w_{Dice} = 0.5, w_{Focal} = 0$
- Imbalanced dataset: $w_{BCE} = 0.2, w_{Dice} = 0.3, w_{Focal} = 0.5$
- Fine boundaries: $w_{BCE} = 0.3, w_{Dice} = 0.7, w_{Focal} = 0$

Training Loop Algorithm

Algorithm 11: Complete Training Loop

Data: Model $\mathcal{M}$, Training data $\mathcal{D}_{train}$, Validation data $\mathcal{D}_{val}$
Result: Trained model parameters θ^*
Initialize:
$\theta \leftarrow$ Xavier_Init() `// Initialize model parameters`
Learning rate η, Epochs E, Batch size B optimizer $\leftarrow$ Adam(θ, η);
scheduler $\leftarrow$ ReduceLROnPlateau(optimizer, patience $= 5$);
best_iou $\leftarrow 0$;
for $e = 1$ *to* E **do**
  Training Phase:
  $\mathcal{M}$.train();
  for $(\mathbf{X}, \mathbf{Y}) \in \mathcal{D}_{train}$ **do**
    $\hat{\mathbf{Y}} \leftarrow \mathcal{M}(\mathbf{X})$;
    $\mathcal{L} \leftarrow$ CombinedLoss$(\hat{\mathbf{Y}}, \mathbf{Y})$;
    `// Backward pass`
    optimizer.zero_grad();
    $\mathcal{L}$.backward();
    `// Gradient clipping (optional)`
    clip_grad_norm$(\theta, \text{max_norm} = 1.0)$;
    optimizer.step();
  end
  Validation Phase:
  $\mathcal{M}$.eval();
  val_iou $\leftarrow$ Evaluate$(\mathcal{M}, \mathcal{D}_{val})$;
  `// Learning rate scheduling`
  scheduler.step(val_loss);
  if *val_iou* $>$ *best_iou* **then**
    best_iou $\leftarrow$ val_iou;
    $\theta^* \leftarrow \theta$ `// Save best model`
  end
end
return θ^*;

IoU Calculation Algorithm

Algorithm 12: Intersection over Union Metric

Data: Predictions $\hat{\mathbf{Y}} \in \mathbb{R}^{B\times C\times H\times W}$, Targets $\mathbf{Y} \in \{0,1\}^{B\times C\times H\times W}$, Threshold $\tau = 0.5$
Result: IoU score $\in [0,1]$

Binary IoU Calculation:
// Convert logits to probabilities
$\mathbf{P} \leftarrow \text{Sigmoid}(\hat{\mathbf{Y}})$;

// Apply threshold
$\mathbf{P}_{binary} \leftarrow (\mathbf{P} > \tau)$;

// Calculate intersection and union
$\text{TP} \leftarrow \sum(\mathbf{P}_{binary} \wedge \mathbf{Y})$ // True Positives
$\text{FP} \leftarrow \sum(\mathbf{P}_{binary} \wedge \neg\mathbf{Y})$ // False Positives
$\text{FN} \leftarrow \sum(\neg\mathbf{P}_{binary} \wedge \mathbf{Y})$ // False Negatives

$\text{intersection} \leftarrow \text{TP}$;
$\text{union} \leftarrow \text{TP} + \text{FP} + \text{FN}$;

$\text{IoU} \leftarrow \frac{\text{intersection}+\epsilon}{\text{union}+\epsilon}$;

Mean IoU for Multi-class:
$\text{mIoU} \leftarrow 0$;
for $c = 1$ *to* C **do**
  $\text{IoU}_c \leftarrow \text{BinaryIoU}(\hat{\mathbf{Y}}[:,c,:,:], \mathbf{Y}[:,c,:,:])$;
  $\text{mIoU} \leftarrow \text{mIoU} + \text{IoU}_c$;
end
$\text{mIoU} \leftarrow \text{mIoU}/C$;

return mIoU;

Memory-Efficient Training with Gradient Accumulation

Algorithm 13: Memory-Efficient Training via Gradient Accumulation

Data: Model $\mathcal{M}$, Data loader $\mathcal{D}$, Effective batch size B_{eff}, Mini-batch size B_{mini}, Accumulation steps $N_{acc} = B_{eff}/B_{mini}$

Gradient Accumulation Training:

optimizer.zero_grad();

step_count $\leftarrow 0$;

for $(\mathbf{X}, \mathbf{Y}) \in \mathcal{D}$ **do**

  // Forward pass with mini-batch

  $\hat{\mathbf{Y}} \leftarrow \mathcal{M}(\mathbf{X})$;

  $\mathcal{L} \leftarrow \text{Loss}(\hat{\mathbf{Y}}, \mathbf{Y})$;

  // Scale loss by accumulation steps

  $\mathcal{L} \leftarrow \mathcal{L}/N_{acc}$;

  // Backward pass (accumulate gradients)

  $\mathcal{L}$.backward();

  step_count $\leftarrow$ step_count $+ 1$;

  if *step_count* mod $N_{acc} = 0$ **then**

    // Update weights after accumulation

    optimizer.step();

    optimizer.zero_grad();

  end

end

When to Use

- GPU memory limited but large batch size needed for stability
- Example: Effective batch = 32, but GPU fits only 8 $\rightarrow$ accumulate 4 steps
- Trade-off: More computation time for same effective batch size

Ablation Study Algorithm

Algorithm 14: Systematic Ablation Study

Data: Base configuration $\mathcal{C}_{base}$, Components to ablate $\{c_1, c_2, ..., c_n\}$
Result: Performance comparison table

Initialize Results:
results $\leftarrow \{\}$;

Baseline Model:
$\mathcal{M}_{base} \leftarrow$ BuildModel($\mathcal{C}_{base}$);
results["baseline"] $\leftarrow$ TrainAndEvaluate($\mathcal{M}_{base}$);

Ablation Experiments:
for *each component* c_i *in* $\{c_1, c_2, ..., c_n\}$ **do**
  $\mathcal{C}_{ablated} \leftarrow \mathcal{C}_{base}$;
  `// Remove or modify component`
  switch c_i **do**
    case *"skip_connections"* **do**
      $\mathcal{C}_{ablated}$.skip $\leftarrow$ False;
    end
    case *"batch_norm"* **do**
      $\mathcal{C}_{ablated}$.use_bn $\leftarrow$ False;
    end
    case *"depth"* **do**
      $\mathcal{C}_{ablated}$.layers $\leftarrow \mathcal{C}_{base}$.layers $- 1$;
    end
    case *"attention"* **do**
      $\mathcal{C}_{ablated}$.skip_mode $\leftarrow$ "concat";
    end
  end
  $\mathcal{M}_{ablated} \leftarrow$ BuildModel($\mathcal{C}_{ablated}$);
  results[c_i] $\leftarrow$ TrainAndEvaluate($\mathcal{M}_{ablated}$);
end

return results;

Common Pitfalls and Solutions

Debugging Checklist

Use this when things don't work as expected!

Dimension Mismatches

Problem: Concatenation fails

- Check padding in convolutions
- Verify upsampling scale matches pooling
- Use `print(x.shape)` liberally

Memory Issues

Problem: CUDA out of memory

- Reduce batch size
- Use gradient accumulation
- Clear cache: `torch.cuda.empty_cache()`
- Use mixed precision training

Poor Convergence

Problem: Loss plateaus early

- Check data normalization
- Verify loss function implementation
- Try gradient clipping
- Reduce learning rate

Overfitting

Problem: Train IoU ¿¿ Val IoU

- Increase dropout
- Add more augmentation
- Use early stopping
- Reduce model capacity

Resources and References

Code Resources

- Starter code: `lab04_student.py`
- Dataset loader: `data_utils.py`
- Evaluation metrics: `metrics.py`

Key Papers to Reference

- Ronneberger et al. (2015): "U-Net: Convolutional Networks for Biomedical Image Segmentation"
- Oktay et al. (2018): "Attention U-Net: Learning Where to Look for the Pancreas"
- Zhou et al. (2018): "UNet++: A Nested U-Net Architecture"

Submission Requirements

- Completed notebook with all tasks
- Training logs and loss curves
- Best model checkpoint (.pth file)
- Brief report (2-3 pages) analyzing results
- **Due**: One week from today