Show and Segment: Universal Medical Image Segmentation via In-Context Learning

Technical Reproduction Guide for Texas Tech HPC

CS Graduate Research Team

August 4, 2025

1 Executive Summary

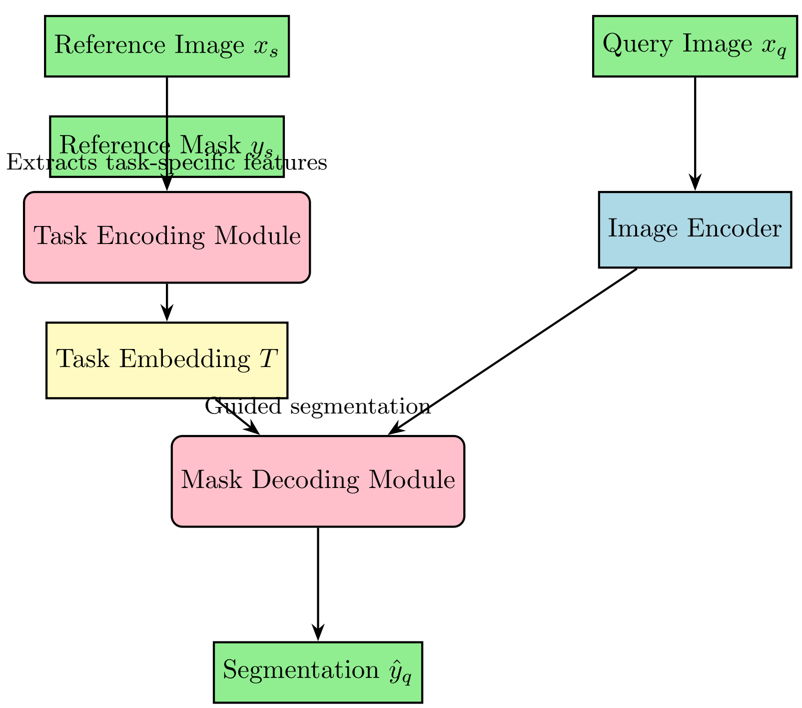
This document provides a comprehensive guide for reproducing the Iris framework proposed by Gao et al. (2025) in "Show and Segment: Universal Medical Image Segmentation via In-Context Learning". The paper introduces a novel approach that enables medical image segmentation on previously unseen tasks without retraining, using reference image-label pairs to guide segmentation.

1.1 Key Contributions

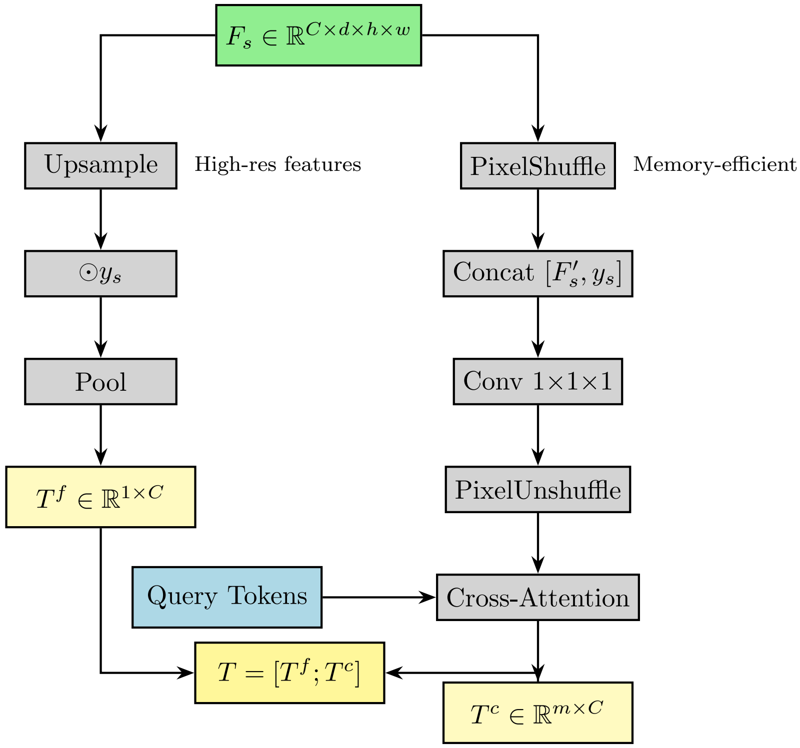
- **In-context learning** for 3D medical images without fine-tuning
- **Lightweight task encoding module** that captures task-specific information
- **Flexible inference strategies** including one-shot, ensemble, and retrieval
- **State-of-the-art performance** on 19 datasets with superior generalization

2 Technical Architecture Overview

2.1 High-Level Architecture Diagram



2.2 Task Encoding Module Architecture



3 Key Technical Terms and Concepts

3.1 In-Context Learning (ICL)

- **Definition**: Ability to perform new tasks using example input-output pairs without updating model parameters
- **In Iris**: Uses reference image-label pairs to define segmentation tasks dynamically
- **Advantage**: Eliminates need for task-specific training or fine-tuning

3.2 Task Encoding

- **Foreground Feature Encoding**: Captures fine boundary details at high resolution
- **Contextual Feature Encoding**: Extracts global context using learnable query tokens
- **Task Embedding**: Compact representation $T \in \mathbb{R}^{(m+1)\times C}$ guiding segmentation

3.3 Inference Strategies

1. **One-shot Inference**: Single reference example
2. **Context Ensemble**: Averages embeddings from multiple references
3. **Object-level Retrieval**: Matches individual anatomical structures
4. **In-context Tuning**: Optimizes task embeddings for specific cases

4 Implementation Guide for Texas Tech HPC

4.1 Environment Setup

Listing 1: SLURM Job Script Template

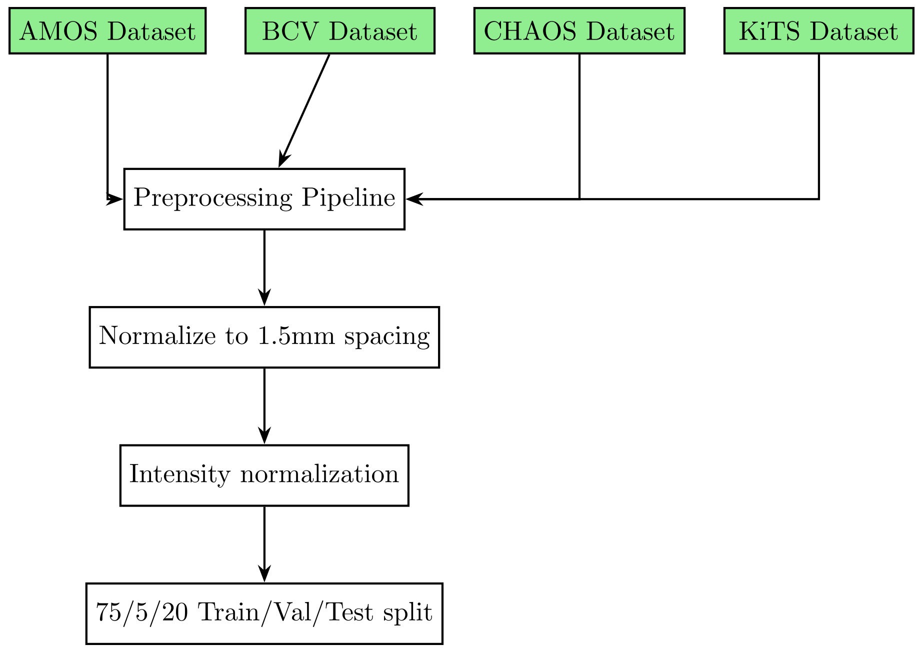
```
#!/bin/bash
#SBATCH --job-name=iris_medical_seg
#SBATCH --output=iris_%j.out
#SBATCH --error=iris_%j.err
#SBATCH --partition=gpu
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=8
#SBATCH --gres=gpu:a100:1
#SBATCH --mem=64GB
#SBATCH --time=48:00:00

# Load modules
module load python/3.9
module load cuda/11.7
module load pytorch/2.0

# Setup environment
conda activate iris_env

# Run training
python train_iris.py --config configs/iris_config.yaml
```

4.2 Python Environment

Listing 2: Conda Environment Setup

```
# Create environment
conda create -n iris_env python=3.9
conda activate iris_env

# Install PyTorch with CUDA support
conda install pytorch==2.0.0 torchvision==0.15.0 pytorch-cuda=11.7 -c pytorch -c nvidia

# Install medical imaging libraries
pip install SimpleITK==2.2.1
pip install nibabel==5.1.0
pip install monai==1.2.0
pip install medicaltorch==0.2

# Install other dependencies
pip install numpy scipy pandas scikit-learn
pip install tqdm tensorboard wandb
pip install pyyaml hydra-core
```

4.3 Data Preparation

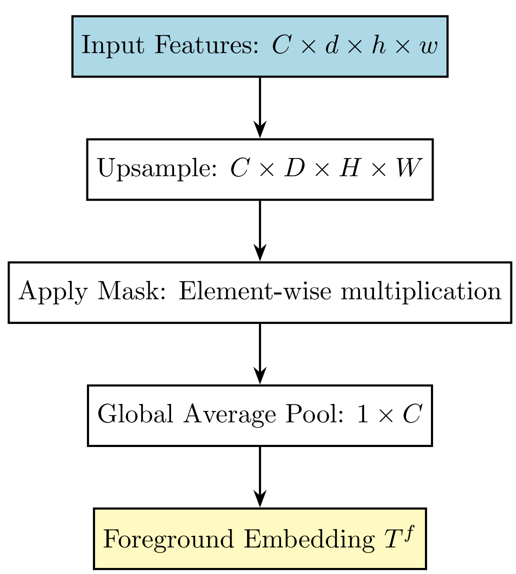


5 Model Architecture Details

5.1 3D UNet Backbone

- **Encoder**: 4 downsampling stages, base channels = 32
- **Decoder**: Symmetric upsampling with skip connections
- **Residual blocks**: Each stage uses residual connections
- **Normalization**: Instance normalization throughout

5.2 Task Encoding Components



6 Training Strategy

6.1 Episodic Training Algorithm

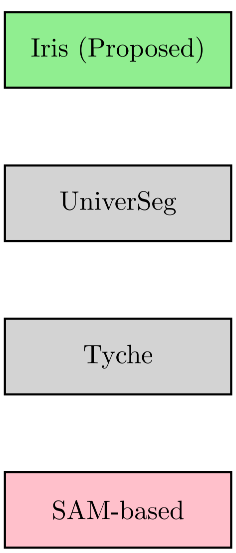
Algorithm 1 Iris Episodic Training

Input: Dataset $D = \bigcup_{k=1}^{K} D_k$
Initialize: Model parameters θ
while not converged **do**
  for $b = 1$ to batch_size **do**
    Sample dataset $k \sim \text{Uniform}(1, K)$
    Sample query pair $(x_q, y_q) \sim D_k$
    Sample reference pair $(x_s, y_s) \sim D_k$
    Add to batch B
  end for
  Extract task embeddings: $T = \text{TaskEncode}(x_s, y_s)$
  Predict: $\hat{y}_q = \text{Decode}(x_q, T)$
  Loss: $L = L_{\text{dice}}(\hat{y}_q, y_q) + L_{\text{ce}}(\hat{y}_q, y_q)$
  Update θ via gradient descent
end while

6.2 Loss Functions

- **Dice Loss**: $L_{\text{dice}} = 1 - \frac{2\sum \hat{y}\cdot y}{\sum \hat{y} + \sum y}$
- **Cross-Entropy Loss**: $L_{\text{ce}} = -\sum y \log(\hat{y})$
- **Combined**: $L = L_{\text{dice}} + L_{\text{ce}}$

7 Comparison with Baseline Methods



Property	In-Context	3D Support	Multi-class
Iris	✓	✓	✓
UniverSeg	✓	×	×
Tyche	✓	×	×
SAM-based	×	Partial	×

8 External Resources and Code

8.1 Official Implementations

- **Iris**: Not yet released (as of paper publication)

- **UniverSeg**: `https://github.com/JJGO/UniverSeg`
- **Tyche**: `https://github.com/mariannerakic/Tyche`
- **nnUNet**: `https://github.com/MIC-DKFZ/nnUNet`

8.2 Related Repositories

- **MONAI**: `https://github.com/Project-MONAI/MONAI`
- **MedicalNet**: `https://github.com/Tencent/MedicalNet`
- **3D-UNet PyTorch**: `https://github.com/wolny/pytorch-3dunet`

8.3 Dataset Access

- **AMOS**: `https://amos22.grand-challenge.org/`
- **Medical Segmentation Decathlon**: `http://medicaldecathlon.com/`
- **KiTS19**: `https://kits19.grand-challenge.org/`

9 Reproduction Steps

9.1 Step 1: Data Download and Organization

Listing 3: Data Organization Structure

```
iris_reproduction/
|-- data/
|   |-- amos/
|   |   |-- imagesTr/
|   |   |-- labelsTr/
|   |   `-- dataset.json
|   |-- bcv/
|   |-- chaos/
|   `-- kits/
|-- src/
|   |-- models/
|   |-- datasets/
|   `-- utils/
`-- configs/
```

9.2 Step 2: Implement Core Components

Listing 4: Task Encoding Module Skeleton

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class TaskEncodingModule(nn.Module):
    def __init__(self, in_channels, embed_dim, num_tokens=10):
        super().__init__()
        self.in_channels = in_channels
        self.embed_dim = embed_dim
        self.num_tokens = num_tokens

        # Foreground encoding components
        self.upsample = nn.Upsample(scale_factor=2, mode='trilinear')
```

```python
        # Context encoding components
        self.pixel_shuffle = PixelShuffle3D(2)
        self.conv1x1 = nn.Conv3d(in_channels//8 + 1, in_channels, 1)
        self.pixel_unshuffle = PixelUnshuffle3D(2)

        # Query tokens and attention
        self.query_tokens = nn.Parameter(torch.randn(num_tokens, embed_dim))
        self.cross_attention = nn.MultiheadAttention(embed_dim, num_heads=8)

    def forward(self, features, mask):
        # Foreground path
        features_up = self.upsample(features)
        foreground = features_up * mask
        tf = F.adaptive_avg_pool3d(foreground, 1).squeeze()

        # Context path
        features_ps = self.pixel_shuffle(features)
        features_concat = torch.cat([features_ps, mask], dim=1)
        features_conv = self.conv1x1(features_concat)
        features_pus = self.pixel_unshuffle(features_conv)

        # Cross-attention with query tokens
        tc = self.cross_attention(self.query_tokens, features_pus)

        # Combine embeddings
        task_embedding = torch.cat([tf.unsqueeze(0), tc], dim=0)
        return task_embedding
```

9.3 Step 3: Training Configuration

Listing 5: Training Configuration YAML

```yaml
# configs/iris_config.yaml
model:
  name: Iris
  encoder:
    channels: [32, 64, 128, 256, 512]
    num_stages: 4
  task_encoder:
    embed_dim: 512
    num_tokens: 10
  decoder:
    num_classes: 1  # Binary segmentation

training:
  batch_size: 4
  num_epochs: 300
  learning_rate: 2e-3
  weight_decay: 1e-5
  optimizer: LAMB
  scheduler:
    name: exponential
    gamma: 0.99

data:
  volume_size: [128, 128, 128]
  spacing: [1.5, 1.5, 1.5]
  augmentation:
    random_crop: true
    random_flip: true
    random_rotate: 10
    intensity_shift: 0.1
```

10 Performance Optimization for HPC

10.1 Multi-GPU Training

Listing 6: Distributed Training Setup

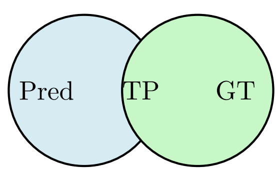
```
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

def setup_distributed(rank, world_size):
    os.environ['MASTER_ADDR'] = 'localhost'
    os.environ['MASTER_PORT'] = '12355'
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def train_distributed(rank, world_size):
    setup_distributed(rank, world_size)

    # Create model and move to GPU
    model = Iris().to(rank)
    model = DDP(model, device_ids=[rank])

    # Create distributed sampler
    train_sampler = DistributedSampler(train_dataset)
    train_loader = DataLoader(
        train_dataset,
        batch_size=batch_size // world_size,
        sampler=train_sampler
    )
```

10.2 Memory Optimization

- **Gradient Checkpointing**: Reduce memory usage during backpropagation
- **Mixed Precision Training**: Use FP16 with automatic mixed precision
- **Sliding Window Inference**: Process large volumes in patches

11 Evaluation Metrics

11.1 Dice Score Calculation



$$\text{Dice} = \frac{2 \times |Pred \cap GT|}{|Pred| + |GT|}$$

12 Troubleshooting Common Issues

12.1 Memory Issues

- Reduce batch size

- Enable gradient checkpointing
- Use smaller volume patches during training

12.2 Convergence Problems

- Verify data preprocessing pipeline
- Check learning rate scheduling
- Ensure proper data augmentation

13 Expected Results

Based on the paper, you should expect:

- **In-distribution Dice**: ∼89.56% average across 12 datasets
- **OOD generalization**: 82-86% on held-out datasets
- **Novel class adaptation**: 28-69% with single reference
- **Training time**: ∼48-72 hours on single A100 GPU

14 Conclusion

This guide provides the technical foundation for reproducing the Iris framework. Key advantages over existing methods include:

1. True 3D volumetric processing (unlike UniverSeg/Tyche)
2. Multi-class segmentation in single forward pass
3. Efficient task encoding reusable across queries
4. Flexible inference strategies for different use cases

For successful reproduction, focus on:

- Careful implementation of the task encoding module
- Proper episodic training setup
- Comprehensive data preprocessing pipeline
- Utilizing HPC resources for multi-GPU training