ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

Machine Learning for Computer Vision

TD2: Image segmentation

Département d'informatique

Ruiwen HE

Course Review: Image segmentation

1. **What is Image Segmentation?**
   Image segmentation is the process of dividing an image into multiple meaningful regions. It plays a crucial role in computer vision by aiding in object recognition and image content understanding. It is widely applied in fields such as medical imaging, autonomous driving, and object detection.

2. **Types of Image Segmentation Tasks:**
   - **Semantic Segmentation:** Classifies each pixel into a category without distinguishing between instances.
   - **Instance Segmentation:** Differentiates between instances of the same class.
   - **Panoptic Segmentation:** A combination of semantic and instance segmentation.

3. **Traditional Image Segmentation Techniques:**
   - **Threshold-based Methods:** Separates foreground and background based on pixel intensity.
   - **Edge-based Methods:** Detect object boundaries (e.g., Sobel, Canny edge detection).
   - **Clustering-based Methods:** Segments by grouping similar pixels (e.g., K-Means, Gaussian Mixture Model).
   - **Region-based Methods:** Groups neighboring pixels that share similar characteristics (e.g., Watershed Algorithm).

4. **Deep Learning-based Image Segmentation:**
   - **Fully Convolutional Networks (FCNs):** Used for semantic segmentation.
   - **U-Net:** Commonly used in medical imaging for precise localization.
   - **Mask R-CNN:** Used for instance segmentation by detecting objects and generating pixel-level masks.
   - **Panoptic FPN:** Combines semantic and instance segmentation into one framework.

5. **Evaluation Metrics for Image Segmentation:**
   - **Pixel Accuracy:** Ratio of correctly classified pixels.
   - **Intersection over Union (IoU):** Measures overlap between predicted and ground truth segmentation.
   - **Dice Coefficient:** Similar to IoU, but assigns more weight to correctly classified pixels.

6. **Applications of Image Segmentation:**
   - **Robotics:** Used in machine vision for object detection and navigation.
   - **Medical Imaging:** Helps in disease diagnosis, organ segmentation, and surgical video annotation.
   - **Autonomous Vehicles:** Critical for object detection and scene understanding.
   - **Satellite Imaging:** Used to segment roads, buildings, and trees in aerial images.

Part I: Basics of Image Segmentation and Evaluation Metrics

Objective:

- Understand the fundamental concepts of image segmentation.
- Learn the types of segmentation tasks and evaluation metrics.

Exercise 1:Implement a function to calculate IoU and Dice Coefficient

- Intersection over Union (IoU): Measures the overlap between predicted segmentation and ground truth.

$$\text{IoU} = \frac{|A \cap B|}{|A \cup B|} = \frac{\sum_{i=1}^{N} y_i \hat{y}_i}{\sum_{i=1}^{N} (y_i + \hat{y}_i) - \sum_{i=1}^{N} y_i \hat{y}_i}$$

  - y_i is the ground truth value for pixel i.
  - $\hat{y}_i$ is the predicted value for pixel i.
  - N is the total number of pixels.

- Dice Coefficient: A measure of overlap between predicted and actual segmentation.

$$\text{Dice} = \frac{2|A \cap B|}{|A| + |B|} = \frac{2\sum_{i=1}^{N} y_i \hat{y}_i}{\sum_{i=1}^{N} y_i + \sum_{i=1}^{N} \hat{y}_i}$$

  - y_i is the ground truth value for pixel i.
  - $\hat{y}_i$ is the predicted value for pixel i.
  - N is the total number of pixels.

- **Implementation**: Create two functions to calculate the Intersection over Union and the Dice Coefficient. Provide execution on examples of your choice. (hint: use libraries like numpy)
- **Question**: How pixel accuracy can be misleading in imbalanced datasets ? (e.g., when the background dominates)
- **Bonus**: Illustrate this with sample data.

Part II: Traditional Image Segmentation Techniques

Objective:

- Learn and implement traditional image segmentation techniques.
- Conduct a comparative analysis of these methods, highlighting their advantages and limitations.

Task:

- Implement Otsu's method for thresholding.
- Use Sobel and Canny edge detection for edge-based segmentation.
- Implement K-Means clustering for image segmentation.
- Use the Watershed algorithm for segmentation.

Exercise 2: Otsu Thresholding

- **Steps**:

1. **Read the Image**
   Load the input image and convert it to grayscale if it is a color image:

$$I_{\text{gray}} = \text{cv2.cvtColor}(I_{\text{image}}, \text{cv2.COLOR\_BGR2GRAY})$$

2. **Compute Histogram**
   Calculate the histogram of the grayscale image to get the pixel intensity distribution:

$$\text{hist} = \text{cv2.calcHist}([I_{\text{gray}}], [0], None, [256], [0, 256])$$

3. **Otsu's Method**
   Apply Otsu's method to determine the optimal threshold value T that minimizes the within-class variance:

$$T = \text{cv2.threshold}(I_{\text{gray}}, 0, 255, \text{cv2.THRESH\_OTSU})[0]$$

4. **Thresholding**
   Create a binary image by applying the threshold T:

$$I_{\text{binary}} = \begin{cases} 255, & \text{if } I_{\text{gray}} > T \\ 0, & \text{if } I_{\text{gray}} \leq T \end{cases}$$

5. **Display Results**
   Show the original grayscale image and the binary image using a plotting library.

- **Implementation**: Implement the previous 5 steps. (hint: use matplotlib for visualization)
- **Question**: What are the strengths and limitations of Otsu's method ?

Exercise 3: Sobel Edge Detection

- **Steps**:

1. **Grayscale Conversion**
   If the input image is a color image (RGB), convert it to grayscale, since edge detection is typically applied to single-channel images. The grayscale image I_{gray} can be computed as:

$$I_{\text{gray}} = 0.2989\, I_R + 0.5870\, I_G + 0.1140\, I_B$$

   where I_R, I_G, and I_B are the red, green, and blue channel intensities, respectively.

2. **Sobel Kernels**
   Define the Sobel kernels for both horizontal and vertical gradients. These kernels are convolved with the grayscale image.
   - **Horizontal Kernel (G_x)**:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

   - **Vertical Kernel (G_y)**:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

3. **Convolution**
   Perform convolution of the Sobel kernels with the grayscale image to compute the gradients in both horizontal (G_x) and vertical (G_y) directions:

$$G_x = I_{\text{gray}} * G_x \quad \text{and} \quad G_y = I_{\text{gray}} * G_y$$

   where $*$ represents the convolution operation.

4. **Gradient Magnitude**
   Compute the magnitude of the gradient at each pixel by combining the horizontal and vertical gradients:
   $$G = \sqrt{G_x^2 + G_y^2}$$
5. **Thresholding (optional)**
   Apply a threshold to the gradient magnitude to create a binary edge map. Pixels with values above the threshold T are considered edges:
   $$G_{\text{thresholded}} = \begin{cases} 1, & \text{if } G > T \\ 0, & \text{if } G \leq T \end{cases}$$

- **Implementation**: Implement the previous 5 steps. (hint: use opencv Sobel and threshold functions, use matplotlib for visualization)
- **Question**: What are the strengths and limitations of Sobel's method ?

Exercise 4: Canny Edge Detection

- **Steps**:

1. **Grayscale Conversion**
   If the input image is a color image (RGB), convert it to grayscale, as Canny edge detection is performed on single-channel images. The grayscale image I_{gray} can be computed as:
   $$I_{\text{gray}} = 0.2989\, I_R + 0.5870\, I_G + 0.1140\, I_B$$
2. **Gaussian Blurring**
   Apply a Gaussian filter to the grayscale image to reduce noise and avoid false edge detection:
   $$I_{\text{blurred}} = G_\sigma * I_{\text{gray}}$$
   where G_σ is the Gaussian kernel.
3. **Gradient Calculation**
   Compute the gradients using Sobel operators to find the intensity and direction of edges:
   $$G_x = I_{\text{blurred}} * G_x \quad \text{and} \quad G_y = I_{\text{blurred}} * G_y$$
4. **Non-Maximum Suppression**
   Thin the edges by suppressing non-maximum values in the gradient direction. This step helps to maintain only the strongest edges.
5. **Double Thresholding**
   Apply two thresholds (low and high) to determine strong and weak edges:
   $$\text{Strong edges} \Rightarrow G > T_{\text{high}} \text{Weak edges} \Rightarrow T_{\text{low}} < G \leq T_{\text{high}}$$
6. **Edge Tracking by Hysteresis**
   Finalize the edges by suppressing weak edges that are not connected to strong edges.

- **Implentation**: Provide an implementation for previous steps. (hint: use opencv Canny function)
- **Questions**: What are the strengths and limitations of Canny Edge method ?

Exercise 5: K-Means Clustering for Image Segmentation

- **Steps**:

1. **Read and Reshape the Image**
   Load the input image and reshape it into a two-dimensional array where each pixel is represented by its RGB values:
   $$\text{pixels} = I_{\text{image}}.reshape(-1, 3)$$
2. **Apply K-Means Clustering**

Use the K-Means algorithm to cluster the pixel values into k clusters:

$$kmeans = KMeans(n\_clusters = k, random\_state = 42)$$

Fit the K-Means model to the pixel data:

$$kmeans.fit(pixels)$$

3. **Replace Each Pixel with its Cluster Centroid**
   Replace each pixel in the image with the centroid of its assigned cluster:

   $$\text{segmented\_image} = kmeans.cluster\_centers_[kmeans.labels\_].reshape(I_{\text{image}}.shape).astype(np.uint8)$$

4. **Display Results**
   Show the original image and the segmented image using a plotting library.

- **Implementation**: Provide an implementation for previous steps.
- **Questions**: What are the advantages and disadvantages of using K-Means for image segmentation compared to other methods? Name at least one other clustering method and compare with K-Means.

Exercise 6: Watershed Algorithm for Image Segmentation

- **Steps**:

1. **Read and Preprocess the Image**
   Load the image and convert it to grayscale if it is in color. Apply Gaussian smoothing to reduce noise:

   $$I_{\text{gray}} = \text{cv2.cvtColor}(I, \text{cv2.COLOR\_BGR2GRAY})$$

   $$I_{\text{smoothed}} = \text{cv2.GaussianBlur}(I_{\text{gray}}, (5,5), 0)$$

2. **Compute the Gradient**
   Compute the gradient of the image to identify the edges:

   $$I_{\text{gradient}} = \text{cv2.Laplacian}(I_{\text{smoothed}}, \text{cv2.CV\_64F})$$

3. **Thresholding and Dilation**
   Apply a binary threshold to the gradient image and perform morphological dilation to enhance the markers:

   $$\text{ret}, \text{markers} = \text{cv2.threshold}(I_{\text{gradient}}, T, 255, \text{cv2.THRESH\_BINARY\_INV})$$

   $$\text{markers} = \text{cv2.dilate}(\text{markers}, \text{None}, \text{iterations} = 3)$$

4. **Watershed Algorithm**
   Apply the watershed algorithm to segment the image:

   $$I_{\text{markers}} = \text{cv2.watershed}(I, \text{markers})$$

   Mark the boundaries of the segmented regions:

   $$I[I_{\text{markers}} == -1] = [0, 0, 255] \quad \text{(Red for boundaries)}$$

5. **Display Results**
   Show the segmented image with the boundaries:

   $$\text{plt.imshow}(I)$$

- **Implementation**: Provide an implementation for previous steps. (hint: use opencv functions GaussianBlur, Laplacian, threshold, dilate and watershed)
- **Question**: What are the advantages and limitations of the watershed algorithm in image segmentation?

Part III: Deep Learning-Based Image Segmentation Methods

Objective:

- Implement various deep learning models for semantic segmentation, instance segmentation, and panoptic segmentation.

Exercise 7: Image Segmentation using Pre-trained FCN Model

- **Steps**:

1. **Import Libraries**
   Import the necessary libraries for image processing and deep learning:

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt
from torchvision import models
import torch
from torchvision import transforms
```

2. **Load the Pre-trained FCN Model**
   Load a pre-trained FCN model from PyTorch's torchvision library:

```python
# Load pre-trained FCN model
model = models.segmentation.fcn_resnet101(pretrained=True)
model.eval()  # Set the model to evaluation mode
```

3. **Preprocess the Input Image**
   Load and preprocess the input image, resizing it and normalizing:

```python
# Load and preprocess the image
image = cv2.imread('image.jpg')
image_resized = cv2.resize(image, (640, 480))  # Resize to match the input size
input_tensor = transforms.ToTensor()(image_resized).unsqueeze(0)  # Add batch dimension
```

4. **Perform Segmentation**
   Use the model to predict segmentation:

```python
# Perform segmentation
with torch.no_grad():
    output = model(input_tensor)['out'][0]  # Get the output from the model
    output_predictions = output.argmax(0)  # Get the predicted class for each pixel
```

5. **Visualize the Segmentation Result**
   Display the original image and the segmentation result:

```python
# Visualize the results
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.title('Original Image')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(output_predictions.cpu().numpy(), cmap='jet', alpha=0.5)
plt.title('Segmentation Result')
plt.axis('off')

plt.show()
```

- **Question**: What are the advantages of using pre-trained models for image segmentation tasks?

Exercise 8: U-Net for Image Segmentation

- **Overview**: U-Net is a convolutional neural network designed for biomedical image segmentation. It employs a symmetric architecture that enables precise localization while maintaining context through skip connections.
- **Architecture**:
  - U-Net consists of a contracting path (encoder) to capture context and a symmetric expanding path (decoder) for precise localization.
  - The network uses skip connections to combine features from the encoder with the decoder, enhancing the model's ability to segment images accurately.
- **Code Example**:

```python
import torch
import torchvision.transforms as transforms
from torchvision import models
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

# Load the pre-trained U-Net model
model = models.segmentation.unet(pretrained=True)
model.eval()

# Load and preprocess the image
image = Image.open('image.jpg').convert('RGB')
transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
input_tensor = transform(image).unsqueeze(0)  # Add batch dimension

# Perform inference
with torch.no_grad():
    output = model(input_tensor)['out'][0]  # Get the output
    output_predictions = output.argmax(0)  # Get the class with the highest score

# Display the results
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(image)

plt.subplot(1, 2, 2)
plt.title("U-Net Segmentation")
plt.imshow(output_predictions.cpu().numpy(), cmap='jet')
plt.colorbar()
plt.show()
```

- **Code Explanation**:
  - **U-Net Model**: This code example demonstrates how to load a pre-trained U-Net model. Note that PyTorch's 'torchvision' library may not directly provide U-Net, so you might need to use alternative implementations like 'segmentation_models.pytorch'.
  - **Image Preprocessing**: The input image is resized to 256×256 pixels and normalized.
  - **Inference**: The model is used to perform inference, and the predicted output is obtained.
  - **Results Display**: The original image and its corresponding segmentation result are displayed.
- **Question**: What are the advantages of using U-Net over traditional segmentation methods, and how do skip connections enhance performance?

Exercise 9: Mask R-CNN for Instance Segmentation

- **Overview**: Mask R-CNN is an extension of Faster R-CNN that adds a branch for predicting segmentation masks on each Region of Interest (RoI), in parallel with the existing branch for classification and bounding box regression.
- **Architecture**:
  - Mask R-CNN consists of a backbone network (such as ResNet) for feature extraction, a Region Proposal Network (RPN) for proposing candidate object bounding boxes, and a branch for predicting segmentation masks.
  - The network uses a Fully Convolutional Network (FCN) for mask prediction, allowing it to output high-resolution segmentation masks.
- **Code Example**:

```python
import torch
from torchvision import models
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import torchvision.transforms as T

# Load the pre-trained Mask R-CNN model
model = models.detection.maskrcnn_resnet50_fpn(pretrained=True)
model.eval()

# Load and preprocess the image
image = Image.open('image.jpg').convert('RGB')
transform = T.Compose([
    T.ToTensor(),
])
input_tensor = transform(image)  # Convert the image to a tensor

# Perform inference
with torch.no_grad():
    predictions = model([input_tensor])  # Get predictions

# Visualize the results
plt.imshow(image)
ax = plt.gca()

# Draw bounding boxes and masks
for i in range(len(predictions[0]['boxes'])):
    box = predictions[0]['boxes'][i].numpy()
    score = predictions[0]['scores'][i].item()
    mask = predictions[0]['masks'][i, 0].mul(255).byte().numpy()

    if score > 0.5:  # Confidence threshold
        ax.add_patch(plt.Rectangle((box[0], box[1]), box[2]-box[0], box[3]-box[1],
                                   fill=False, edgecolor='red', linewidth=2))
        plt.imshow(mask, alpha=0.5)  # Overlay mask

plt.title("Mask R-CNN Segmentation")
plt.axis('off')
plt.show()
```

- **Code Explanation**:
  - **Mask R-CNN Model**: This code snippet demonstrates how to load a pre-trained Mask R-CNN model using PyTorch's 'torchvision' library.

- **Image Preprocessing**: The input image is transformed into a tensor suitable for the model.
- **Inference**: The model performs inference on the input image, outputting bounding boxes, scores, and masks for detected objects.
- **Results Visualization**: The original image is displayed with overlaid bounding boxes and masks for detected objects, filtered by a confidence threshold.

- **Question**: What are the benefits of using Mask R-CNN for instance segmentation compared to traditional methods, and how does it manage the trade-off between speed and accuracy?

Exercise 10: Panoptic FPN Segmentation

- **Overview**: The Panoptic Feature Pyramid Network (Panoptic FPN) is designed to achieve high-quality object detection and segmentation by leveraging multi-scale feature representations. This approach performs both instance segmentation and semantic segmentation in a unified framework.

- **Steps**:

1. **Load Pre-trained Model**
   Import the necessary libraries and load a pre-trained Panoptic FPN model, typically trained on the COCO dataset.
2. **Prepare Input Image**
   Preprocess the input image to match the expected dimensions and normalize pixel values. The input should be a tensor with shape $[1, 3, H, W]$.
3. **Perform Inference**
   Pass the input image through the model to obtain predictions, including bounding boxes, segmentation masks, and class labels.
4. **Post-Processing**
   Apply post-processing steps to refine the predictions, such as thresholding for confidence scores and applying non-maximum suppression to eliminate duplicate boxes.
5. **Visualize Results**
   Visualize the segmentation results by overlaying the predicted masks and bounding boxes on the original image.

- **Code Example**:

```python
import torch
from torchvision import models
import torchvision.transforms as T
from PIL import Image
import matplotlib.pyplot as plt

# Load the pre-trained Panoptic FPN model
model = models.detection.maskrcnn_resnet50_fpn(pretrained=True)
model.eval()

# Load and preprocess the input image
input_image = Image.open('path/to/your/image.jpg').convert("RGB")
transform = T.Compose([
    T.Resize((800, 800)),
    T.ToTensor(),
])
input_tensor = transform(input_image).unsqueeze(0)  # Add batch
    dimension

# Perform inference
with torch.no_grad():
    predictions = model(input_tensor)

# Visualize the results
plt.figure(figsize=(12, 8))
plt.imshow(input_image)
```

```
26
27 # Overlay masks and boxes
28 for i in range(len(predictions[0]['boxes'])):
29     box = predictions[0]['boxes'][i].cpu().numpy()
30     score = predictions[0]['scores'][i].cpu().numpy()
31     mask = predictions[0]['masks'][i, 0].cpu().numpy()
32
33     if score > 0.5:  # Confidence threshold
34         plt.imshow(mask, alpha=0.5)  # Overlay mask
35         plt.gca().add_patch(plt.Rectangle(
36             (box[0], box[1]), box[2]-box[0], box[3]-box[1],
37             fill=False, color='red', linewidth=2
38         ))
39
40 plt.axis('off')
41 plt.title("Panoptic FPN Segmentation Results")
42 plt.show()
```

- **Code Explanation**:
  - The code imports necessary libraries, including PyTorch and torchvision, for loading and using the pre-trained model.
  - It loads the pre-trained Panoptic FPN model using 'maskrcnn_resnet50_fpn', which combines object detection and instance segmentation.
  - An input image is loaded and preprocessed, resized to 800×800 pixels and converted to a tensor.
  - The model performs inference on the input tensor, providing predictions that include bounding boxes, segmentation masks, and confidence scores.
  - The results are visualized by overlaying the predicted masks and bounding boxes onto the original image, applying a confidence threshold to filter out less certain predictions.
- **Question**: What are the key differences between FPN and the two previous methods ?

Part IV: Training on Your Own Dataset (BONUS)

Objective:
Learn how to prepare a custom dataset for image segmentation and train a deep learning model (e.g., U-Net or Mask R-CNN) on this dataset.
Content Overview:

- **Dataset Preparation:**
  - How to format your dataset for training.
  - Creating images and corresponding masks.
  - Splitting the dataset into training and validation sets.
- **Model Setup:**
  - Choosing a model architecture (e.g., U-Net, Mask R-CNN).
  - Configuring hyperparameters.
- **Training the Model:**
  - Setting up the training loop.
  - Monitoring training and validation metrics.
- **Evaluating Results:**
  - Evaluating the trained model on the test set.
  - Visualizing the predictions.