CS 6410 Final Project
May 2024
Group members: Mansour Rezaei, Daniel Safavisohi

Outlines:

- Introduction
    - History of image segmentation and related works
    - More on final project
- First experiment
- Second experiment
- Results
- Conclusion

Introduction:
History of image segmentation and related works

In the dynamic field of image segmentation, evolving techniques have significantly refined the process of dividing images into meaningful segments, known as semantic segmentation. This involves assigning each pixel in an image to a specific category that represents an object or boundary. Traditionally, simpler methods like thresholding and graph theory were used, but they lacked precision and the ability to operate at the pixel level.

With the advent of deep learning, particularly Convolutional Neural Networks (CNNs), the landscape has shifted dramatically. CNNs leverage spatial contextual information without the need for handcrafted features, simplifying the segmentation process and enhancing accuracy. This advancement has spawned several robust models such as U-shaped Networks (UNet), which excels in medical imaging by using fewer training samples effectively, and Fully Convolutional Networks (FCNs), which adapt to any image size for accurate pixel-wise predictions.

Moreover, models like DeepLab and PSPNet have integrated advanced concepts like atrous convolution and pyramid pooling to handle multi-scale segmentation tasks more efficiently. The integration of probabilistic graphical models with CNNs, such as CRFasRNN, further refines segmentation by combining deep learning predictions with refined boundary delineation.

More on final project

In this project, we utilized the VOC 2007 dataset to train a U-Net model specifically tailored for image segmentation tasks. The aim was to effectively segment images into 22 predefined classes using the robust architecture of the U-Net model.

The VOC 2007 dataset, a standard benchmark in image segmentation, was downloaded from its official website as a tar file. We structured the dataset into three subsets: training, validation, and testing, with the respective ratios of 8:1:1. This split ensured a comprehensive training process while allowing for effective model validation and testing. Preprocessing steps included resizing the images to 224x224 pixels and converting them to grayscale to reduce computational complexity. Corresponding masks were also prepared and assigned to 22 different classes, reflecting the diverse categories present in the VOC dataset.

First Experiment

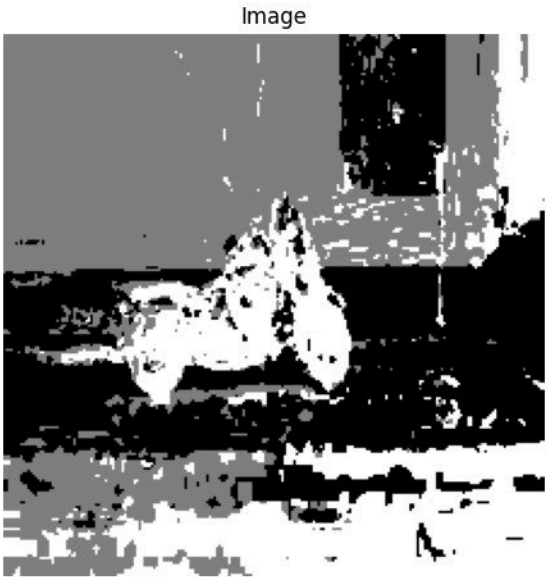
In experiment 1, we used cmap function from the gist provided (https://gist.github.com/wllhf/a4533e0adebe57e3ed06d4b50c8419ae) :

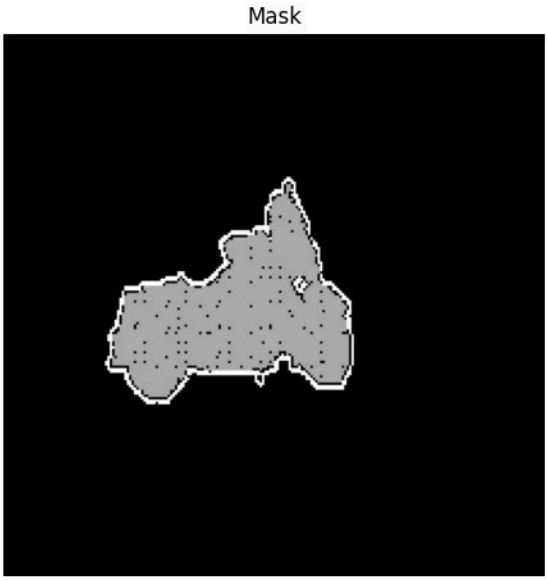
```python
import numpy as np
from skimage.io import imshow
import matplotlib.pyplot as plt

from skimage.transform import resize
from skimage.io import imread
import os

def color_map(N=256, normalized=True):
    def bitget(byteval, idx):
        return ((byteval & (1 << idx)) != 0)

    dtype = 'float32' if normalized else 'uint8'
    cmap = np.zeros((N, 3), dtype=dtype)
    for i in range(N):
        r = g = b = 0
        c = i
        for j in range(8):
            r = r | (bitget(c, 0) << 7-j)
            g = g | (bitget(c, 1) << 7-j)
            b = b | (bitget(c, 2) << 7-j)
            c = c >> 3

        cmap[i] = np.array([r, g, b])

    cmap = cmap/255 if normalized else cmap
    return cmap
```

And we decided to define preprocessing function as this:

```python
def preprocess_and_save(image_path, mask_path, cmap, output_size=(224, 224), output_dir='output', log = False):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    # Load and resize image
    # image = imread(image_path)

    # # Load and resize mask
    # mask_image = imread(mask_path)

    image = imageio.imread(image_path)
    mask_image = imageio.imread(mask_path)
    if log:
        plot_sample(image, mask_image, title=f"{image_path}")

    mask_image = resize(mask_image, output_size, anti_aliasing=False, preserve_range=True).astype(int)
    image = resize(image, output_size, anti_aliasing=True) / 255.0  # Normalize to [0, 1]


    # Map color to channel index
    color_to_index = {tuple(val): idx for idx, val in enumerate(cmap)}

    # Initialize 22-channel binary mask
    channels = np.zeros((*output_size, 22), dtype=np.float32)

    # Vectorized mask processing
    # for idx, val in enumerate(cmap):
    #     channels[:, :, idx] = np.all(mask_image == val, axis=-1)

    channels = np.zeros((*output_size, 22), dtype=np.float32)  # Assuming 22 classes including background
    for idx, color in enumerate(cmap):
        mask = np.all(mask_image == np.array(color*256, dtype=int), axis=-1).astype(int)
        if log:
            print(mask)
        channels[:, :, idx] = mask
    if log:
        plot_sample(image, channels, title=f"Sample")
        # Use this in your preprocessing function to log unique colors of some masks
        print("Unique colors in mask:", unique_colors(mask_image))

    # Save image and mask to binary files
    image_filename = os.path.join(output_dir, os.path.basename(image_path) + '_image.npy')
    mask_filename = os.path.join(output_dir, os.path.basename(mask_path) + '_mask.npy')
    np.save(image_filename, image)
    np.save(mask_filename, channels)
```

Then we check the mask with the corresponding image in gray scale:





Next, we define the dice loss function as follows:

```python
# Define Dice loss function and coefficient
def dice_coefficient(y_true, y_pred, smooth=1e-6):
    y_true_f = tf.keras.backend.flatten(y_true)
    y_pred_f = tf.keras.backend.flatten(y_pred)
    intersection = tf.keras.backend.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (tf.keras.backend.sum(y_true_f) + tf.keras.backend.sum(y_pred_f) + smooth)

def dice_loss(y_true, y_pred):
    return 1 - dice_coefficient(y_true, y_pred)
```

Finally, we designed U-net model and train the model on training data:

```python
def unet_model(input_size=(224, 224, 3), num_classes=22):
    inputs = Input(input_size)

    # Downward path
    c1 = Conv2D(32, (3, 3), activation='relu', padding='same')(inputs)
    c1 = Dropout(0.1)(c1)
    c1 = Conv2D(32, (3, 3), activation='relu', padding='same')(c1)
    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(64, (3, 3), activation='relu', padding='same')(p1)
    c2 = Dropout(0.1)(c2)
    c2 = Conv2D(64, (3, 3), activation='relu', padding='same')(c2)
    p2 = MaxPooling2D((2, 2))(c2)

    # Bottleneck
    c3 = Conv2D(128, (3, 3), activation='relu', padding='same')(p2)
    c3 = Dropout(0.2)(c3)
    c3 = Conv2D(128, (3, 3), activation='relu', padding='same')(c3)

    # Upward path
    u4 = UpSampling2D((2, 2))(c3)
    u4 = concatenate([u4, c2])
    c4 = Conv2D(64, (3, 3), activation='relu', padding='same')(u4)
    c4 = Dropout(0.1)(c4)
    c4 = Conv2D(64, (3, 3), activation='relu', padding='same')(c4)

    u5 = UpSampling2D((2, 2))(c4)
    u5 = concatenate([u5, c1])
    c5 = Conv2D(32, (3, 3), activation='relu', padding='same')(u5)
    c5 = Dropout(0.1)(c5)
    c5 = Conv2D(32, (3, 3), activation='relu', padding='same')(c5)

    outputs = Conv2D(num_classes, (1, 1), activation='sigmoid')(c5)

    model = Model(inputs=[inputs], outputs=[outputs])
    model.compile(optimizer=Adam(learning_rate=1e-5), loss=dice_loss, metrics=[dice_coefficient])

    return model

# Initialize the U-Net model
model = unet_model()
model.summary()  # This will print the summary of the model without needing Graphviz
```

It turns out out that the model doesn't work properly, and we got nan loss function with very low dice coefficient that doesn't improve and worth reporting.

Second Experiment

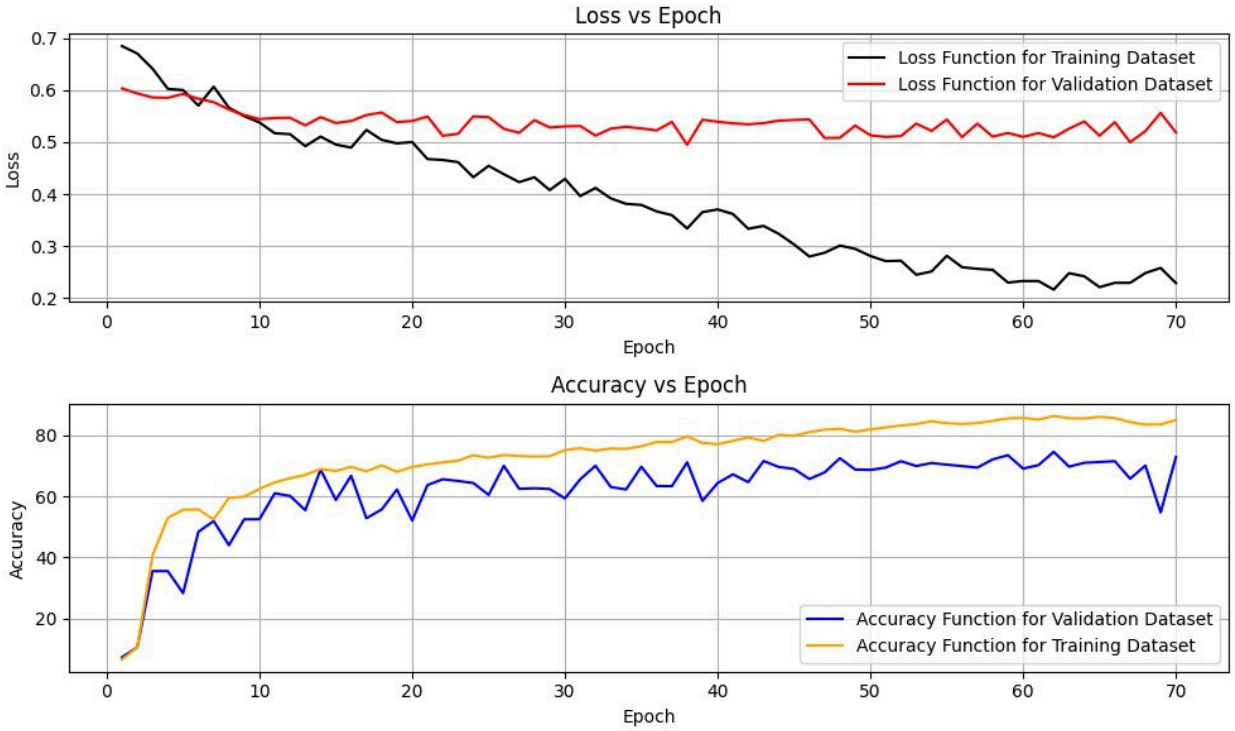
In our next experiment, with ResNet structure in mind, we construct a U-net model with 16 Conv2D layers, 16 BatchNorm2D layars, 3 BasicBlock layers and 1 MaxPooling layer. We decided to use Relu activation function for each layer and Softmax for the last layer. We used Dice loss function as advised.

The U-Net model was trained on the prepared training dataset using a batch size of 64 and learning rate of 0.0005. The training process was conducted over 70 epochs, with performance monitored through the Dice loss metric. Validation was intermittently performed to adjust parameters and prevent overfitting. We trained the model on GPU Nvidia GeForce TRX 3080, 32 GB Ram and CPU AMD Ryzen 7.
Below is a snippet from the model used in our second experiment:

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 112, 112]	9,408
BatchNorm2d-2	[-1, 64, 112, 112]	128
ReLU-3	[-1, 64, 112, 112]	0
MaxPool2d-4	[-1, 64, 56, 56]	0
Conv2d-5	[-1, 64, 56, 56]	36,864
BatchNorm2d-6	[-1, 64, 56, 56]	128
ReLU-7	[-1, 64, 56, 56]	0
Conv2d-8	[-1, 64, 56, 56]	36,864
BatchNorm2d-9	[-1, 64, 56, 56]	128
ReLU-10	[-1, 64, 56, 56]	0
BasicBlock-11	[-1, 64, 56, 56]	0
Conv2d-12	[-1, 64, 56, 56]	36,864
BatchNorm2d-13	[-1, 64, 56, 56]	128
ReLU-14	[-1, 64, 56, 56]	0
Conv2d-15	[-1, 64, 56, 56]	36,864
BatchNorm2d-16	[-1, 64, 56, 56]	128
ReLU-17	[-1, 64, 56, 56]	0
BasicBlock-18	[-1, 64, 56, 56]	0
Conv2d-19	[-1, 64, 56, 56]	36,864
BatchNorm2d-20	[-1, 64, 56, 56]	128
ReLU-21	[-1, 64, 56, 56]	0
Conv2d-22	[-1, 64, 56, 56]	36,864
BatchNorm2d-23	[-1, 64, 56, 56]	128
ReLU-24	[-1, 64, 56, 56]	0
BasicBlock-25	[-1, 64, 56, 56]	0
Conv2d-26	[-1, 128, 28, 28]	73,728
BatchNorm2d-27	[-1, 128, 28, 28]	256
ReLU-28	[-1, 128, 28, 28]	0
Conv2d-29	[-1, 128, 28, 28]	147,456
BatchNorm2d-30	[-1, 128, 28, 28]	256
Conv2d-31	[-1, 128, 28, 28]	8,192
BatchNorm2d-32	[-1, 128, 28, 28]	256

We decided to record log data in a text file to be able to use it for further analysis. The model's performance was continuously evaluated on the validation set during training. We could not run Tensorboard as it is not compatible with wsl2, so we decided to record loss metrics for each epoch and visualize the learning progress and convergence behavior using matplotlib.

Upon completion of the training phase, the model was tested on the unseen data of the test dataset which includes 10 percent of the original dataset. This step was crucial to assess the generalizability of the U-Net model outside the training data.
Below is the graph of loss function and accuracy versus number of epochs:



The model achieved a very good accuracy of 75.1% for validation datastet, which reveals its effectiveness in handling image segmentation tasks.

At the end, we apply model on test dataset and compare the results:

In the above graph, images on the first, second and third row are real one, prediction and ground truth, respectively. This comparison reveals that our model has done a good job, but still far from prefect class segmentation.

Conclusion:
We believe that this project was really challenging and we need more time to come up with a better model. Having said that, the results of our model shows capability of U-net model on image segmentation. You can find our codes and complementary explanations in our GitHub page (https://github.com/toldo4/cs_6410_final_project/tree/master).

References:

1. Jiao, Libin, Lianzhi Huo, Changmiao Hu, and Ping Tang. 2020. "Refined UNet: UNet-Based Refinement Network for Cloud and Shadow Precise Segmentation" *Remote Sensing* 12, no. 12: 2001. https://doi.org/10.3390/rs12122001
2. Sun, Yu, Fukun Bi, Yangte Gao, Liang Chen, and Suting Feng. 2022. "A Multi-Attention UNet for Semantic Segmentation in Remote Sensing Images" *Symmetry* 14, no. 5: 906. https://doi.org/10.3390/sym14050906
3. Y. Weng, T. Zhou, Y. Li and X. Qiu, "NAS-Unet: Neural Architecture Search for Medical Image Segmentation," in *IEEE Access*, vol. 7, pp. 44247-44257, 2019, doi: 10.1109/ACCESS.2019.2908991.
4. Fu, Leiyang, and Shaowen Li. 2023. "A New Semantic Segmentation Framework Based on UNet" *Sensors* 23, no. 19: 8123. https://doi.org/10.3390/s23198123
5. Quan Zhou, Xiaofu Wu, Suofei Zhang, Bin Kang, Zongyuan Ge, Longin Jan Latecki, Contextual ensemble network for semantic segmentation,Pattern Recognition,Volume 122,2022,108290,ISSN 0031 3203,https://doi.org/10.1016/j.patcog.2021.108290.
6. Paradis, Frédérik, David Beauchemin, Mathieu Godbout, Mathieu Alain, Nicolas Garneau, Stefan Otte, Alexis Tremblay, Marc-Antoine Bélanger, and François Laviolette. 2020. *Poutyne: A Simplified Framework for Deep Learning*. Accessed [date]. https://poutyne.org.
7. Tensorflow Documentation, URL=https://www.tensorflow.org/api_docs.
8. PyTorch Documentation, URL=https://pytorch.org/docs.
9. NumPy Documentation, URL=https://numpy.org/doc.
10. Matplotlib Documentation, URL=https://matplotlib.org.
11. Kaggle Notebooks, URL=https://www.kaggle.com/docs/notebooks.