Task 1 - DL - Eyeglasses Segmentation Report

The objective is to develop and train a deep learning model to accurately segment eyeglasses in facial images. You will be provided with a dataset of facial images containing annotations for the eyeglasses region. You have the freedom to choose any suitable deep learning architecture for image segmentation. After training, evaluate the model's performance with corresponding metrics and perform benchmarks.

Figure 1. Training data example

Note: If you don't have a CUDA-capable device, use GoogleColab for training.

Deliverable:

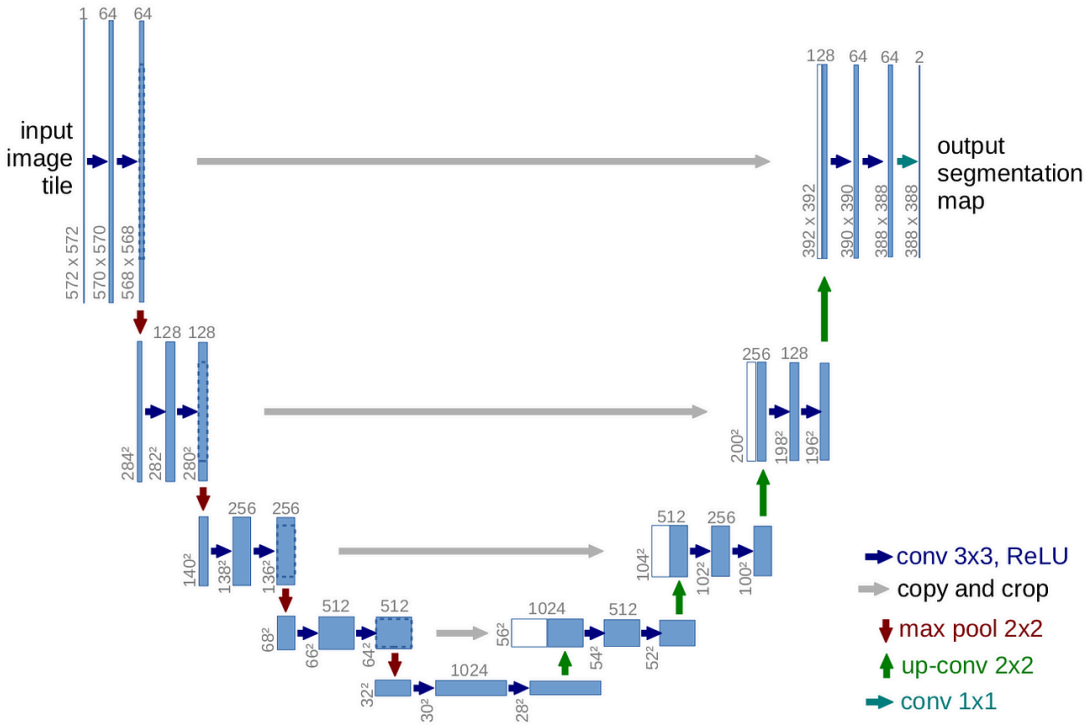
- Training code & inference code (`.py` or `.ipynb`).
- Saved model (a format compatible with your chosen DL framework).
- Visualization of segmentation results on particular images from the `test` folder.
- A concise report (`.pdf` or `.ipynb` format) documenting the training process, including model architecture details, training configuration, evaluation metrics and their achieved values, benchmarks and any relevant visualizations.

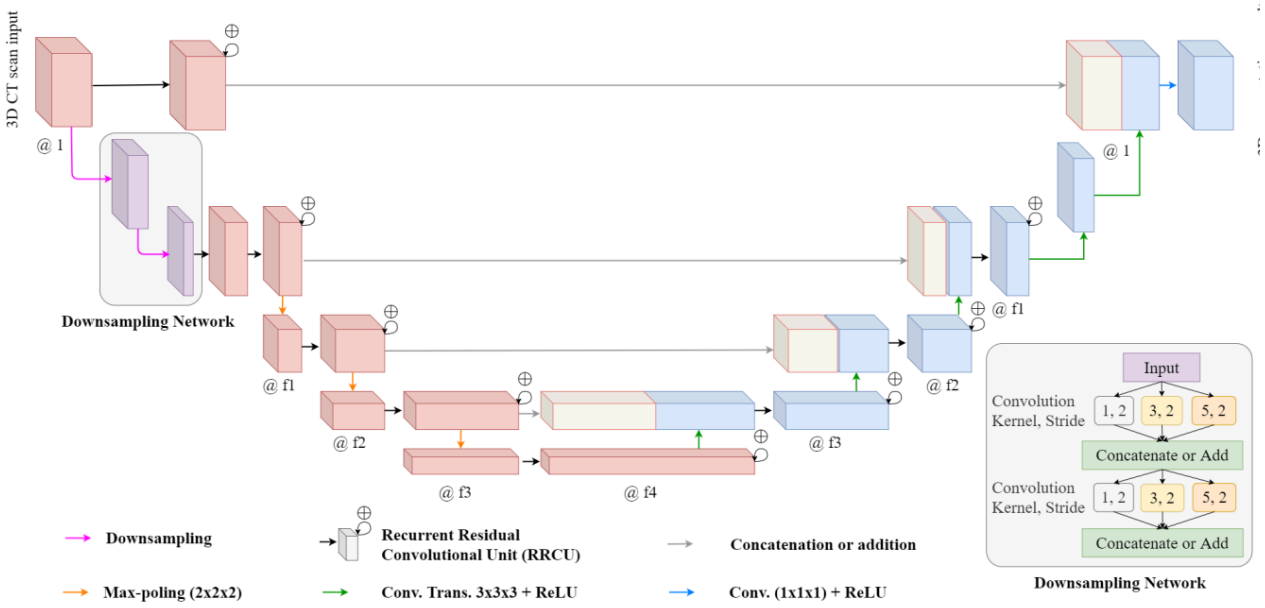
Bonus Task [not mandatory]:
Convert the trained model to a CPU-compatible format and demonstrate its quality and speed performance by running inference on test images using the CPU.

Introduction

In this work, I have chosen to employ a well-established and classical approach, which is particularly well-suited for tasks such as semantic segmentation. Drawing on my experience, I opted for the classical U-Net architecture, renowned for its efficacy in medical image analysis and its robust performance in capturing intricate spatial information. The U-Net's ability to effectively handle diverse segmentation challenges, thanks to its encoder-decoder structure and skip

connections, makes it an ideal chose for this particular task. By leveraging this architecture, I aim to achieve high accuracy in pixel-level classification while preserving the spatial integrity of the segmented regions.
It is an ideal choice for this task as it does not require extensive computational resources for training and allows for a straightforward, yet powerful, architectural design. For this task also suit models like: SegFormer, SETR, Segment Anything etc…, but i decided focus on something already know.
Below, i provide a visualization from U-Net paper, it's exactly how i implemented model in this task.





U-Net consists of two main parts:

1. **Contracting Path (Encoder)**: This part is responsible for capturing context and features from the input image.
2. **Expanding Path (Decoder)**: This part is used to enable precise localization and upsampling to generate the segmentation mask.

Encoder

The contracting path follows a typical architecture of a convolutional network. It comprises multiple convolutional layers and pooling operations. The main goal is to extract high-level features while reducing the spatial dimensions of the input.

1. **Convolutional Layers**:
    - Each step in the contracting path consists of two convolutional layers with 3x3 filters.
    - These layers are followed by a ReLU activation function.
2. **Pooling Layers**:
    - After the convolutions, a 2x2 max pooling operation is applied.
    - Pooling reduces the spatial dimensions by half, allowing the network to gain spatial invariance and reduce computation.
3. **Feature Maps**:
    - The number of feature maps is doubled after each pooling operation, allowing the network to learn more complex features at each stage. Number of feature maps: [3, 64, 126. 256. 512]

Bottleneck

Between the contracting and expanding paths lies the bottleneck, which serves as a bridge. It consists of two 3x3 convolutional layers followed by ReLU activations, without any pooling. The bottleneck captures the most abstract and high-level features of the input.

Decoder

The expanding path mirrors the contracting path but with the addition of upsampling layers. The main function of this path is to upsample the feature maps back to the original input size to generate the segmentation mask.

1. **Up-convolutional (Transposed Convolutional) Layers**:
    - Each step in the expanding path begins with a 2x2 up-convolution (transposed convolution) that increases the spatial dimensions by a factor of two.
    - This step halves the number of feature maps.
2. **Concatenation**:
    - The upsampled feature map is concatenated with the corresponding feature map from the contracting path.

    - This allows the network to combine high-resolution features from the contracting path with the upsampled features, preserving spatial context and details.
3. **Convolutional Layers**:
    - Two 3x3 convolutional layers followed by ReLU activations are applied after the concatenation.
    - This step refines the features and improves the spatial resolution.

Final step

The final layer of the U-Net architecture features a 1x1 convolution, elegantly reducing the number of feature maps to match the number of desired output classes, such as the background and the object class. To conclude the process, a sigmoid activation function is typically employed, ensuring the generation of probability scores for each pixel, gracefully transforming logits into meaningful segmentation outputs.

Training stage

During the training process, I carefully selected the AdamW optimizer to ensure efficient and effective weight adjustments. The model was trained using a batch size of 10 and was subjected to 30 epochs to ensure thorough learning. Each input image had a resolution of 512x512 pixels, which provided a detailed level of granularity. The learning rate was set at 1e-4, striking a balance between convergence speed and stability. The training was conducted on a powerful NVIDIA RTX 3090 GPU with 24GB of VRAM, enabling swift and parallel computations.

For the data preprocessing, I employed a straightforward transformation technique, using only resizing to match the input dimensions, maintaining the simplicity and efficiency of the preprocessing pipeline. The loss function used was Binary Cross-Entropy with Logits (BCEWithLogitsLoss), which is well-suited for binary classification tasks, effectively handling the logistic regression of each pixel.

To measure the performance of the model, I tracked two critical metrics: the Dice coefficient and binary accuracy. The Dice coefficient provided an insightful measure of the overlap between the predicted and ground truth masks, offering a clear indication of segmentation quality. Meanwhile, binary accuracy was used to evaluate the proportion of correctly classified pixels, ensuring that the model's predictions were both accurate and reliable.

Training process:

```
Epoch [1/26], Loss: 0.3273, Metrics: {'dice coefficient': -0.18486858148127794, 'binary accuracy': 1.9073486612342096e-09}
Epoch [2/26], Loss: 0.2023, Metrics: {'dice coefficient': -0.21643397908657788, 'binary accuracy': 0.0}
Epoch [3/26], Loss: 0.1459, Metrics: {'dice coefficient': -0.22009203590452672, 'binary accuracy': 1.9073486612342096e-09}
Epoch [4/26], Loss: 0.1085, Metrics: {'dice coefficient': -0.20091591961681843, 'binary accuracy': 0.0}
Epoch [5/26], Loss: 0.0824, Metrics: {'dice coefficient': -0.186549663618207, 'binary accuracy': 0.0}
Model saved to saved models/best model 0.0690.pth
Epoch [6/26], Loss: 0.0690, Metrics: {'dice coefficient': -0.15716713182628156, 'binary accuracy': 0.0}
Epoch [7/26], Loss: 0.0520, Metrics: {'dice coefficient': -0.15966700837016107, 'binary accuracy': 0.0}
Epoch [8/26], Loss: 0.0421, Metrics: {'dice coefficient': -0.1536177930980921, 'binary accuracy': 1.9073486612342096e-09}
Epoch [9/26], Loss: 0.0364, Metrics: {'dice coefficient': -0.1418813869357109, 'binary accuracy': 0.0}
Epoch [10/26], Loss: 0.0306, Metrics: {'dice coefficient': -0.13848813086748124, 'binary accuracy': 0.0}
Model saved to saved models/best model 0.0279.pth
Epoch [11/26], Loss: 0.0279, Metrics: {'dice coefficient': -0.12643283616751433, 'binary accuracy': 0.0}
Epoch [12/26], Loss: 0.0229, Metrics: {'dice coefficient': -0.1236394988372922, 'binary accuracy': 0.0}
Epoch [13/26], Loss: 0.0190, Metrics: {'dice coefficient': -0.12354046646505594, 'binary accuracy': 0.0}
Epoch [14/26], Loss: 0.0166, Metrics: {'dice coefficient': -0.12142900794744492, 'binary accuracy': 1.9073486612342096e-09}
Epoch [15/26], Loss: 0.0149, Metrics: {'dice coefficient': -0.11889625445008278, 'binary accuracy': 0.0}
Model saved to saved models/best model 0.0133.pth
Epoch [16/26], Loss: 0.0133, Metrics: {'dice coefficient': -0.1165278361365199, 'binary accuracy': 0.0}
Epoch [17/26], Loss: 0.0121, Metrics: {'dice coefficient': -0.11489571306854486, 'binary accuracy': 0.0}
Epoch [18/26], Loss: 0.0113, Metrics: {'dice coefficient': -0.11178145978599786, 'binary accuracy': 0.0}
Epoch [19/26], Loss: 0.0165, Metrics: {'dice coefficient': -0.09192158417776226, 'binary accuracy': 0.0}
Epoch [20/26], Loss: 0.0112, Metrics: {'dice coefficient': -0.0945142888277769, 'binary accuracy': 0.0}
Model saved to saved models/best model 0.0093.pth
Epoch [21/26], Loss: 0.0093, Metrics: {'dice coefficient': -0.09719287216663361, 'binary accuracy': 0.0}
Epoch [22/26], Loss: 0.0085, Metrics: {'dice coefficient': -0.0977180404216051, 'binary accuracy': 0.0}
Epoch [23/26], Loss: 0.0079, Metrics: {'dice coefficient': -0.09794943954795599, 'binary accuracy': 0.0}
Epoch [24/26], Loss: 0.0075, Metrics: {'dice coefficient': -0.09690773122012615, 'binary accuracy': 0.0}
Epoch [25/26], Loss: 0.0072, Metrics: {'dice coefficient': -0.096142000220716, 'binary accuracy': 0.0}
Model saved to saved models/best model 0.0068.pth
```

Testing results:

In conclusion, I want to thank you for developing such an interesting test tasks and I hope that you will give more opportunities to prove yourself as a specialist, thank you in advance.