Open a command prompt or terminal on your computer.

Navigate to the directory where your main.py script is located. You can use the cd (change directory) command to move to the directory. For example, if your main.py is in a folder called "my_project," you would navigate to that directory like this:

```
cd /path/to/your/project/my_project
```

Replace /path/to/your/project/my_project with the actual path to your project directory.

Once you are in the correct directory, you can run the main.py script by using the python command followed by the script's filename. In this case, you would run:

```
python main.py
```

The script will start, and you will see the following prompt:

```
Choose an option (train/predict/exit):
```

You can then follow the instructions provided by the script and enter one of the available options: "train," "predict," or "exit."

U will be asked to enter The values of the hyperparameters
1. batch_size, 2.lr (learning rate), and 3.epochs

Detailed overview of the code

Main.py

1. The script starts with some import statements for modules named **data**, **model**, **train**, and **predict**. These are custom modules that contain functions related to data loading, model building, training, and making predictions.
2. In the **main()** function, there is an infinite loop (**while True**) that allows the user to select from three options: "train," "predict," or "exit."
3. If the user selects "train," the script proceeds to train a U-Net model using some user-specified or default training parameters such as batch size, learning rate, and the number of epochs. It loads the data, prints some data details, creates a model instance, prints the model summary, and then initiates the training process using the **train_unet** function.
4. If the user selects "predict," the script seems to be designed to make predictions on test images. It loads test images from a directory, and the predictions are likely made using the **make_predictions** function.
5. If the user selects "exit," the script breaks out of the loop and terminates.
6. If the user enters an invalid choice, the script informs them and prompts them to enter a valid option.

Data.py

1. **load_data(path, split=0.1)**: This function is responsible for loading and splitting the dataset into training, validation, and test sets. It takes a path to the dataset directory and an optional split ratio as input. It uses the **glob** module to find image and mask files in the specified directories. It splits the data into training, validation, and test sets based on the specified split ratio.
2. **read_image(path)**: This function reads and preprocesses an image given its file path. It uses OpenCV (**cv2**) to read the image, resize it to a specified size (256x256), and normalize the pixel values to be in the range [0, 1].
3. **read_mask(path)**: Similar to **read_image**, this function reads and preprocesses a mask image (typically a grayscale image). It reads the image, resizes it, normalizes the pixel values, and adds an extra dimension to make it compatible with the model input.
4. **tf_parse(x, y)**: This function defines a TensorFlow data parsing function. It uses **tf.numpy_function** to apply **_parse** (the **_parse** function takes individual image and mask paths and returns preprocessed images and masks) to each element (x, y) of the dataset. It sets the shapes for the parsed images and masks.
5. **tf_dataset(x, y, batch=8)**: This function creates a TensorFlow dataset by combining the x and y data, parsing them using **tf_parse**, batching the data into specified batch sizes, and repeating the dataset indefinitely. It's designed to be used for training and can be fed into a machine learning model in TensorFlow.

Model.py

1. **conv_block(x, num_filters)**: This function defines a convolutional block with two consecutive 3x3 convolutional layers, each followed by batch normalization and ReLU activation. It's a basic building block for the encoder and decoder parts of the U-Net.
2. **build_model()**: This function assembles the entire U-Net model. It defines the model architecture with an encoder-decoder structure. The key components include:
   - **size**: The size of the input images (assumed to be 256x256 pixels).
   - **num_filters**: A list of the number of filters for each convolutional layer in the encoder and decoder parts. It starts with decreasing values in the encoder and then reverses the list for the decoder.
   - **inputs**: This defines the input layer for the model with shape (256, 256, 3).

The model architecture consists of:

   - **Encoder**: A series of convolutional blocks with max-pooling layers that reduce the spatial dimensions.
   - **Bridge**: A convolutional block that connects the encoder and decoder parts of the U-Net.
   - **Decoder**: A series of up-sampling layers followed by convolutional blocks. Skip connections from the encoder are used to concatenate feature maps at each decoder level.
   - **Output**: A 1x1 convolutional layer followed by a sigmoid activation function to produce the final segmentation mask.

3. **if __name__ == "__main__"**: This code block ensures that the model is built and summarized only when the script is run directly (not imported as a module). It creates an instance of the U-Net model using the **build_model()** function and prints a summary of the model's architecture.

The U-Net architecture is a popular choice for image segmentation tasks, and this code defines a simple yet effective U-Net model for such tasks. You can use this model for tasks like image segmentation by providing appropriate input data and target labels during training.

Train.py

1. Import TensorFlow and relevant callbacks, as well as custom functions from **data.py** and **model.py**.
2. Define the Intersection over Union (IoU) custom metric. The IoU function is used to measure the similarity between predicted and ground truth masks. It's calculated by finding the intersection and union of the two masks.
3. Define the **train_unet** function, which handles the training process. It takes the data directory, batch size, learning rate, and the number of epochs as input arguments. The steps involved in training are as follows:
    - Load and preprocess data using the **load_data** function from **data.py**.
    - Create TensorFlow datasets for training and validation using the **tf_dataset** function from **data.py**.
    - Build the U-Net model using the **build_model** function from **model.py**.
    - Compile the model with binary cross-entropy loss and a set of metrics (accuracy, recall, precision, and the custom IoU metric).
    - Define a list of callbacks including model checkpointing, learning rate reduction on plateau, CSV logging, TensorBoard logging, and early stopping.
    - Calculate the number of steps per epoch for training and validation datasets.
    - Train the model using the **model.fit** method, specifying training and validation datasets, the number of epochs, steps per epoch, and callbacks.
4. In the **if __name__ == "__main__":** block, hyperparameters such as data directory, batch size, learning rate, and the number of epochs are defined. The **train_unet** function is then called to start the training process.

Predict.py

1. Import necessary libraries, including OpenCV, NumPy, TensorFlow, and custom functions from other modules.
2. Define several utility functions for reading images, reading masks, and parsing masks:
    - **read_image(path)**: Reads and resizes an image.
    - **read_mask(path)**: Reads and resizes a grayscale mask.
    - **mask_parse(mask)**: Transposes and processes a mask for visualization.
3. Define the **make_predictions** function, which is responsible for making predictions on the test dataset and saving the results as images. The steps involved are as follows:
    - Load the test dataset and set the batch size.
    - Determine the number of steps needed for iterating through the test dataset.
    - Load the pre-trained model (presumably trained for image segmentation) using **tf.keras.models.load_model**.
    - Evaluate the model's performance on the test dataset using **model.evaluate**.
    - Iterate through the test images and make predictions using the loaded model.
    - Create visualizations for each test image, including the original image, the ground truth mask, and the predicted mask.
    - Save the visualizations as separate image files in the "results" directory.

4. In the **if __name__ == "__main__":** block, the **make_predictions** function is called with the data directory ("CVC-612/") where the test images and masks are located.