Documentation For the Phase 3 of the Project

Visionary Vanguard

Emad Deilam Salehi – Amin Saeidi

Classifier.py

This Python code is using a pre-trained ResNet-50 model for image classification. It has two main functions: `classifier()` and `preprocessing_and_predict(model, image_bytes)`. This code is included into the project to be sure that the user understand that it should upload a membrane image for more precise results.

The `classifier()` function creates an instance of the ResNet-50 model with two output classes. It loads the weights of a previously trained model from a checkpoint file (`resnet_ft.pt`) and assigns those weights to the model. This file can be downloaded from the google drive link due to its huge size.

The `preprocessing_and_predict(model, image_bytes)` function takes the trained model and an image in the form of bytes as input. It performs the following steps:

1. Sets the model to evaluation mode using `model.eval()`.
2. Opens the image using the PIL library (`from PIL import Image`), converting it to the RGB color mode.
3. Defines a series of image transformations using `transforms.Compose()`. It resizes the image to 224x224 pixels and converts it to a tensor.
4. Applies the defined transformation to the image and adds an extra dimension to create a mini-batch of size 1 using `unsqueeze(0)`.
5. Passes the image tensor through the model to obtain the output logits.
6. Uses `torch.max(output.data, 1)` to find the index of the maximum value along dimension 1 (which represents the predicted class).
7. Prints the predicted class index.
8. Based on the predicted class index, assigns a class name ("CELL MEMBRANE" or "OTHER") to the `class_name` variable.
9. Returns the predicted class index and the corresponding class name.

Note: The model assumes a binary classification task where class 1 represents "CELL MEMBRANE" and class 0 represents "OTHER."

Image_preprocessing.py

This Python code defines a class called "image_preprocessing" that contains several methods for image preprocessing tasks. Let's go through each method and understand what it does:

1. The class initializes with a COLOR_DICT variable, which is a lookup table for colors corresponding to different classes. Each class is represented by an RGB color code.
2. The `adjustData` method takes an image (`img`), a mask (`mask`), a flag indicating whether it's a multi-class segmentation problem (`flag_multi_class`), and the number of classes (`num_class`). It performs data normalization and converts the mask into a one-hot encoded representation. If `flag_multi_class` is true, the mask is converted into a one-hot tensor with a shape of `(height, width, num_class)`. Otherwise, the mask is reshaped into a flattened array with a shape of `(height * width,)` if it's a single-class problem. The image and mask are then returned.
3. The `testGenerator` method generates a test image generator. It takes an image (`img`), target size (`target_size`), a flag indicating multi-class segmentation (`flag_multi_class`), and whether to convert the image to grayscale (`as_gray`). It normalizes the image by dividing it by 255, resizes it to the target size using the `skimage.transform.resize` function, and reshapes it to include a single channel. If `flag_multi_class` is true, the image is returned as is. Otherwise, the image is reshaped to have a shape of `(1, height, width, 1)` and yielded as a generator.
4. The `labelVisualize` method visualizes the labeled image based on the provided number of classes (`num_class`), color dictionary (`color_dict`), and image array (`img`). It assigns colors to each class in the image based on the color dictionary lookup and returns the visualized image.
5. The `saveResult` method saves the segmented result based on the provided NumPy file (`npyfile`). It takes additional parameters such as `flag_multi_class` (indicating multi-class segmentation) and `num_class` (number of classes). It calls the `labelVisualize` method to visualize the segmented image and returns the result.

Overall, the code provides methods for data adjustment, generating test images, visualizing labels, and saving segmentation results.

UNET_tf.py

The code defines a U-Net architecture model using the Keras library for image segmentation tasks. U-Net is a popular convolutional neural network (CNN) architecture commonly used for tasks like medical image segmentation.

Here's a breakdown of the code:

1. The necessary modules are imported from Keras: `Model`, `Input`, `Conv2D`, `MaxPooling2D`, `Dropout`, `UpSampling2D`, and `concatenate`, as well as the `Adam` optimizer.
2. The `unet` function is defined, which takes two optional arguments: `pretrained_weights` and `input_size`. The `pretrained_weights` argument allows the model to be initialized with pre-trained weights if provided, and `input_size` specifies the size of the input image.
3. The input layer is defined with the given `input_size`.
4. The U-Net architecture is built using a series of convolutional and pooling layers. Each block consists of two 3x3 convolutional layers with ReLU activation, followed by a 2x2 max pooling layer. The number of filters in the convolutional layers increases as the network goes deeper, following a pattern of 64, 128, 256, 512, and 1024 filters.
5. A dropout layer is applied with a dropout rate of 0.5 after the fourth convolutional block.

6. The decoding path of the U-Net is constructed using up-convolutional layers (`UpSampling2D`) combined with skip connections from the corresponding encoding layers (`concatenate`). Each up-convolutional block consists of a 2x2 up-convolution layer followed by two 3x3 convolutional layers with ReLU activation.
7. The final layer of the U-Net is a 1x1 convolutional layer with sigmoid activation, which produces a binary mask representing the segmented output.
8. The model is created using the `Model` function from Keras, specifying the input and output layers.
9. The model is compiled with the Adam optimizer using a learning rate of 1e-4 and the binary cross-entropy loss function. The accuracy metric is also specified.
10. If `pretrained_weights` are provided, the model's weights are loaded.
11. Finally, the model is returned.

main.py

This Python code sets up a FastAPI web server for image processing and segmentation using machine learning models. Here is a general overview of what the code does:

1. Import necessary modules and libraries: The code imports various modules and libraries required for image processing, machine learning, and web server functionalities. These include FastAPI, image processing libraries like PIL and numpy, machine learning models, and utility modules.
2. Create FastAPI app: The code creates a FastAPI instance called `app` that represents the web application.
3. Set up CORS middleware: Cross-Origin Resource Sharing (CORS) middleware is added to allow requests from the web page's origin. This enables the web application to make requests to the API.
4. Mount static files: The code mounts the "static" directory to serve static files such as HTML, CSS, and JavaScript files.
5. Define routes and request handlers:
    - The root route ("/") is defined with an HTTP GET method. It reads the content of the "index.html" file located in the "static" directory and returns it as an HTML response.
    - The "/old_model" route is defined with an HTTP POST method. It receives an uploaded image file, preprocesses it, and uses a machine learning classifier model to predict the image class. The predicted image is then encoded as a base64 string and returned as a response along with other metrics such as CPU usage, memory usage, and latency.
    - The "/new_model" route is similar to "/old_model" but uses a different version of the machine learning model.
    - The "/predict" route is defined with an HTTP POST method and randomly routes the request to either "/new_model" or "/old_model" based on a canary percentage. This allows testing and comparison between the two models.
6. Define utility functions:
    - The `reading_image_properly` function reads and preprocesses the uploaded image, checks its format, and uses a classifier model to predict its class. It returns the preprocessed image, a check flag, and the image class name.

- The `preprocess_predict_image` function preprocesses the image and uses a segmentation model (U-Net) to predict a segmentation map. It saves the resulting segmentation map and returns it.
- The `monitor_hardware` function monitors CPU and memory usage using the psutil library and returns the percentages.
- The `measure_latency` function measures the time elapsed since a specified start time and returns the latency in seconds.

7. Run the server: Finally, the code uses the Uvicorn server to run the FastAPI app on the specified host and port (0.0.0.0:8000).

This code deploys a Canary model for image classification and segmentation. The Canary model allows for testing a new version of the machine learning models in a production-like environment before fully rolling it out. In this case, approximately 20% of the incoming requests are randomly routed to the "/new_model" route, while the remaining 80% are routed to the "/old_model" route. This setup enables the evaluation and comparison of the performance and results between the two models.

It is important to note that the optimized model's weights for the U-Net segmentation model are stored in the "unet_membrane.hdf5" file. However, due to the large size of the file, it was not included in the project's GitHub page or the uploaded zip file on Quera site. Therefore, to run the code successfully, you need to obtain the "unet_membrane.hdf5" file separately from the given google drive link.

Overall, this code provides a web API for image classification and segmentation, allowing users to upload images and receive predictions along with associated metrics. The Canary deployment approach ensures controlled testing and comparison between different versions of the machine learning models, while the FastAPI framework facilitates the development of the web server with ease.

Web Page (Static Folder)

This web page is a user interface for a biomedical image segmentation project. It consists of HTML, CSS, and JavaScript code.

The HTML code defines the structure and content of the web page. It contains two main sections: the welcome container and the main container. The welcome container is the initial page displayed to the user and includes a logo, a welcome message, a project overview, and a "Start" button. When the "Start" button is clicked, a JavaScript function called `startProject()` is executed, which applies a fade-out animation to the welcome container and then hides it, revealing the main container.

The main container is where the image segmentation functionality is presented. It includes two image containers: one for the original picture and another for the segmented picture. The user can upload an image using a file input field within a form and click the "Segment Image" button to initiate the image segmentation process. When the button is clicked, a JavaScript function is triggered, which reads the uploaded image file, sends it to a FastAPI web application endpoint (`http://localhost:8000/predict`) using an XMLHttpRequest, and handles the response. The response contains the segmented image data and a classification message. The segmented image is

displayed in the segmented picture container, and the classification message is shown in the result text container.

The CSS code provides custom styles for the web page, defining the layout, colors, and typography. It styles the different containers, image containers, form container, and result container.

The JavaScript code adds interactivity and functionality to the web page. It handles the start button click event to transition from the welcome page to the main container. It also limits the number of words per line in the project overview text by splitting the text and adding line breaks. Additionally, it handles the image upload and segmentation process. When the "Segment Image" button is clicked, it reads the uploaded image file, sends it to the FastAPI endpoint, and updates the web page with the segmented image and classification result.

Overall, this web page provides a user-friendly interface for uploading an image and performing biomedical image segmentation using a FastAPI web application.

The submitted material includes a video titled 'Project Demo.webm', which serves to demonstrate the project's functionality and its successful implementation. Due to the large size of the video, it is uploaded here on the google drive.