Documentation For the Phase One of the Project

Visionary Vanguard

Emad Deilam Salehi – Amin Saeidi

image_preprocessing Class

This Python code defines a class called ***image_preprocessing*** for performing image segmentation tasks. The class has three instance variables ***train_path***, ***test_path***, and ***augmented_path***, which store the paths to the training, testing, and augmented image datasets, respectively.

The class also defines a set of RGB color values for various image segmentation classes such as ***Sky***, ***Building***, ***Pole***, ***Road***, ***Pavement***, ***Tree***, ***SignSymbol***, ***Fence***, ***Car***, ***Pedestrian***, ***Bicyclist***, and ***Unlabelled***. These color values will be used to represent the corresponding segmentation classes in the output images.

The RGB color values are stored in a numpy array called ***COLOR_DICT***, which has dimensions of 12 x 3. Each row of the array represents an RGB color value for a particular segmentation class.

Method 1: __init__

The constructor method ***__init__*** takes three arguments ***train_path***, ***test_path***, and ***augmented_path*** and initializes the corresponding instance variables. It also checks if the ***augmented_path*** directory exists, and if not, it creates a new directory with that name.

Method 2: *adjustData*

The method ***adjustData*** is used to preprocess the input image and mask data before being used for segmentation. The method takes four arguments, ***img***, ***mask***, ***flag_multi_class***, and ***num_class***. The ***img*** argument contains the input image data, and ***mask*** contains the corresponding ground truth segmentation data. The ***flag_multi_class*** argument is a Boolean value that indicates whether the segmentation task is a multi-class or binary classification problem. The ***num_class*** argument specifies the number of segmentation classes.

If ***flag_multi_class*** is true, the method first normalizes the input ***img*** and extracts the segmentation mask for each class from the input ***mask***. It then creates a one-hot encoding for each segmentation class and stores it in a new mask array. Finally, the method reshapes the new mask array and returns the normalized ***img*** and the new mask.

If ***flag_multi_class*** is false, the method first normalizes the input ***img*** and ***mask*** and then converts the ***mask*** data into binary format. Finally, the method returns the normalized ***img*** and binary ***mask***.

Method 3: trainGenerator()

This method generates images and their corresponding masks on-the-fly using Keras ***ImageDataGenerator***. It takes several arguments such as ***batch_size***, ***train_path***, ***image_folder***, ***mask_folder***, ***aug_dict***, ***image_color_mode***, ***mask_color_mode***, ***image_save_prefix***, ***mask_save_prefix***, ***flag_multi_class***, ***num_class***, ***save_to_dir***, ***target_size***, and ***seed***.

batch_size specifies the size of the batch of images and masks to generate. ***train_path*** is the path of the directory containing the training images and masks. ***image_folder*** and ***mask_folder*** are the names of the

folders containing the images and masks, respectively. ***aug_dict*** is a dictionary of data augmentation parameters for the images and masks.

image_color_mode and ***mask_color_mode*** specify the color mode of the input images and masks, respectively. They can be either "grayscale" or "rgb". ***image_save_prefix*** and ***mask_save_prefix*** are the prefix of the filenames of the generated images and masks.

flag_multi_class specifies whether the task is multi-class or not. If ***flag_multi_class*** is True, ***num_class*** specifies the number of classes in the segmentation task. If ***save_to_dir*** is not None, the generated images and masks are saved to the specified directory. ***target_size*** specifies the size of the images and masks after resizing.

seed is used to set the random seed for the image and mask generators, so that the same transformation is applied to both the image and mask during training. Finally, the method returns a generator object that yields batches of images and masks.

Method 4: testGenerator()

This method generates test images on-the-fly. It takes the ***test_path*** of the directory containing the test images, the ***num_image*** of the number of test images, ***target_size*** of the size of the resized images, ***flag_multi_class*** specifying if the task is multi-class or not, and ***as_gray*** specifying if the input image should be converted to grayscale or not.

The method reads in the test images one by one using ***io.imread()*** and resizes them to ***target_size*** using ***trans.resize()***. If ***flag_multi_class*** is False, the image is reshaped into a 3D tensor by adding an extra dimension of size 1 at the end. If ***as_gray*** is True, the image is converted to grayscale. Finally, the method returns a generator object that yields the resized test images.

Method 5: geneTrainNpy()

The method ***geneTrainNpy*** takes in the paths to the image and mask folders, and returns the image and mask arrays in numpy format. It first finds all the image files with the specified prefix in the image folder, then reads in the corresponding mask files with the same prefix in the mask folder. It then applies the ***adjustData*** method to the image and mask data to prepare them for training, and appends them to two separate lists. Finally, it returns the two lists as numpy arrays.

Method 6: labelVisualize()

The method ***labelVisualize*** takes in the number of classes, a dictionary mapping class indices to RGB values, and an image array with shape (height, width, 1) or (height, width) representing the class labels. It generates a color visualization of the class labels by assigning each class a unique RGB color from the dictionary and mapping the class labels to the corresponding colors. It returns the color visualization as an RGB image array with shape (height, width, 3). The color values are divided by 255 to ensure they are in the range [0, 1].

Defining Data Generator

This code block initializes an instance of the ***image_preprocessing*** class and sets the paths for the training and test data, as well as the path for augmented data. It also sets the arguments for data augmentation in

the ***data_gen_args*** dictionary. This dictionary object contains a set of parameters for performing data augmentation and image preprocessing tasks using the TensorFlow Addons (tfa) library. Here's what each parameter does:

- **rotation_range**: Randomly rotates the image by a specified angle (in degrees) within the given range.
- **width_shift_range**: Shifts the image horizontally (left or right) by a fraction of its total width.
- **height_shift_range**: Shifts the image vertically (up or down) by a fraction of its total height.
- **shear_range**: Applies a shear transformation to the image, which distorts its shape along one axis.
- **zoom_range**: Randomly zooms into or out of the image by a specified factor.
- **horizontal_flip**: Flips the image horizontally, which can help to increase the variability of the dataset.
- **preprocessing_function**: Applies a Gaussian filter to the image using the gaussian_filter2d function from the TensorFlow Addons library. A Gaussian filter smooths the image and reduces noise, which can improve the performance of the model.
- **fill_mode**: Specifies the method used to fill in any empty or missing pixels that may result from the data augmentation process. The nearest mode fills in missing pixels with the nearest pixel value.

Next, the code generates a ***myGenerator*** object using the ***trainGenerator*** method of the ***image_preprocessing*** class. This generator object will generate batches of training data with size 20, and will use the training image and label directories, as well as the ***data_gen_args*** dictionary for data augmentation.

Additionally, if ***save_to_dir*** is specified in the ***trainGenerator*** method, the generator will also save augmented images and labels to the directory specified.

Visualizing Data Augmentation Result

This block of code is a loop that generates batches of augmented data using the ***trainGenerator*** method from the ***image_preprocessing*** class. It generates 3 batches of 20 augmented images each (total 60 images) from the ***train_path*** directory and saves them to the ***aug_path*** directory if the ***save_to_dir*** parameter is set to ***aug_path*** in the ***trainGenerator*** method.

Each batch is yielded as a tuple containing two arrays: the first array contains the augmented images and the second array contains their corresponding masks. The loop goes through each batch and breaks after ***num_batch*** (3 in this case) iterations.

Transfer Learning

This code defines a function ***transfer_learning*** (this function is intended for transfer learning, where the pre-trained model is used as a feature extractor and the extracted features are used as input to another model to solve a segmentation task) that takes in a pre-trained model and a parent directory path. The function loops through all the images in the directory, loads each image and preprocesses it to fit the pre-trained model's input dimensions. The preprocessed image is then passed to the pre-trained model for feature extraction, and the extracted features are appended to a numpy array initialized earlier. Finally, the function returns the numpy array containing all the extracted features.

In the last code section, the VGG16 model is loaded with pre-trained weights from ImageNet and its top layer is excluded by setting ***include_top=False***. The ***image*** module from ***keras.preprocessing*** is also imported for loading and preprocessing images. The ***preprocess_input*** function from ***keras.applications.vgg16*** is used for preprocessing the input image data to the VGG16 model.

The function ***transfer_learning*** is defined which takes the VGG16 model and the path to the parent directory of the images as input. It then loads each image from the specified directory, resizes it to 224x224, preprocesses it using ***preprocess_input***, and uses the VGG16 model to extract features from the image. These features are then stored in a numpy array called ***features***. Finally, the shape of ***features*** is printed to the console.

In the main block of the code, the ***transfer_learning*** function is called with the VGG16 model and the parent path of the images. The resulting ***features*** numpy array is printed to the console.