Datasets statisticsc

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| IMAGE NAME | NUMBER OF PIXELS | BIT DEPTH | INTENSITY DISTRIBUTION | LOCATION CAPUTRED (Logtitude,Latitude) |  |
| project\_image01 | 3000 x 4000 | 24 |  | 50.97708587251168, 11.32823229061499 |  |
| project\_image02 | 3000 x 4000 | 24 |  | 50.97710064952938, 11.328354331114184 |  |
| project\_image03 | 3000 x 4000 | 24 |  | 50.97715794175943, 11.328162405449383 |  |
| project\_image04 | 3000 x 4000 | 24 |  | 50.97754996398573, 11.326829455349138 |  |
| project\_image05 | 3000 x 4000 | 24 |  | 50.9773506868792, 11.326901874986023 |  |
| project\_image06 | 3000 x 4000 | 24 |  | 50.98141372110729, 11.325057935521766 |  |
| project\_image07 | 3000 x 4000 | 24 |  | 50.98141372110729, 11.325057935521766 |  |
| project\_image08 | 3000 x 4000 | 24 |  | 50.976367996930556, 11.328694337418222 |  |
| project\_image09 | 3000 x 4000 | 24 |  | 50.97579308445506, 11.329277187257592 |  |
| project\_image010 | 3000 x 4000 | 24 |  | 50.97580585135471, 11.32930084285779 |  |

**1. Data Collection and Preparation:**

* We Gather a large dataset of images that contain cracks. These images should be labeled with pixel-level annotations, with "0" representing no-crack and "255" representing crack.
* Split the dataset into training, validation, and test sets.

**2. Preprocessing:**

* Resize all images to a consistent size.
* Normalize pixel values to a common scale (e.g., [0, 1]).

**3. Model Selection:**

* Choose a neural network architecture suitable for semantic segmentation. RoboFlow is a good choice for this task due to its effectiveness in capturing fine details.
* You can also use pre-trained models for the encoder part (backbone) to improve performance.

**4. Model Design:**

* Design your model to take an image as input and output a segmentation mask with two classes (no-crack and crack).
* The final layer should have a softmax activation function to produce class probabilities for each pixel.
* Use appropriate loss functions, such as categorical cross-entropy or Dice loss, for training.

**5. Data Augmentation:**

* Apply data augmentation techniques like random rotations, flips, and brightness adjustments to increase the diversity of training data and improve the model's robustness.

**6. Training:**

* Train the model on the training dataset using the chosen loss function.
* Monitor the model's performance on the validation set to prevent overfitting.
* Experiment with different learning rates and schedules to find the best training strategy.

**7. Post-processing:**

* After obtaining segmentation results, you may apply post-processing techniques such as morphological operations (e.g., dilation and erosion) to refine the binary masks.

**8. Evaluation:**

* Evaluate the model's performance on the test dataset using appropriate metrics like Intersection over Union (IoU), Dice coefficient, or pixel accuracy.

**9. Fine-tuning (Optional):**

* If the initial model performance is not satisfactory, you can fine-tune the model with additional data or adjust hyperparameters.

**10. Deployment:**

* Once you are satisfied with the model's performance, deploy it for real-world crack detection tasks.

**11. Continuous Improvement:**

* Regularly update the model with new data to adapt to different crack types and environmental conditions.

Task 2

**1. Adaptive Thresholding:**

Adaptive Thresholding calculates a local threshold for each pixel based on the pixel values in its neighborhood. This is particularly useful when dealing with images of varying brightness levels. Here's how to implement adaptive thresholding:

* Choose a suitable neighborhood size (e.g., a small window).
* For each pixel in the image, calculate a local threshold based on the mean or median pixel value in its neighborhood.
* Compare the pixel value to the local threshold and classify it as either crack or no-crack.

Task 2 – Crack Segmentation

In this task we propose an approach to semantically segment the cracks in the image. **Semantic segmentation**

In the task where every pixel in the input image is assigned a class label in the output.

1. Load the image in grayscale mode.
2. Apply thresholding to create a binary image.
3. Invert the binary image.
4. Perform connected component analysis to identify distinct regions in the binary image.
5. Extract various features (area, perimeter, and circularity) for each region.
6. Classify regions as "crack" or "no-crack" based on circularity.
7. Visualize the connected components with a color-coded overlay.
8. Print the labels (crack or no-crack) for each region.

Here's a breakdown of the code:

* The **for** loop iterates from 1 to 27 (inclusive) to process images from "project\_image\_1.jpg" to "project\_image\_27.jpg."
* It loads each image, converts it to grayscale, and applies thresholding with a threshold value of 90. The result is a binary image where pixel values are either 0 (black) or 255 (white).
* The binary image is inverted using **cv2.bitwise\_not** to create a binary image where cracks are represented as white regions on a black background.
* Connected component analysis is performed on the binary image to label and identify distinct regions. The number of labels (**num\_labels**) represents the total number of regions detected.
* For each labeled region (excluding the background label), the code calculates features such as area, perimeter, and circularity. Circularity is a measure of how close a region is to being a perfect circle.
* Based on a circularity threshold of 0.2, regions are classified as either "crack" or "no-crack."
* The code then visualizes the connected components by overlaying color-coded regions on the original image using **matplotlib**.
* Finally, it prints the labels (crack or no-crack) for each detected region in the current image.