Fundamentals of Computer Vision - B. Nasihatkon
Spring 2025

K. N. Toosi
University of Technology

Convolutional Neural Network Applications: Crack Segmentation

Ramin Tavakoli, Mohammad Madi Daghighi, Athar Hakimzadeh
Directed by Araz Abedini Bakhshmand

Crack segmentation plays a pivotal role in infrastructure maintenance. Traditional methods are time-consuming and sometimes hazardous. By leveraging deep learning-based segmentation, authorities can rapidly detect and quantify cracks in images and estimate the required budget.

Image segmentation is a fundamental computer vision task that partitions an image into meaningful regions, assigning each pixel a label. In crack segmentation, the goal is to classify every pixel in an image as either *crack* or *non-crack*, producing a pixel-wise mask that highlights defects.

1. Preparing Dataset

We will start our journey from scratch, so we need to label and prepare our dataset.

1.1. Collect your dataset

To build a robust dataset for crack segmentation, each one of you is required to **collect 40 unique images** of **asphalt cracks** and also **10 undamaged asphalt**. These images will contribute to training a machine learning model, so diversity and quality are essential.

Use your smartphone or camera to take high-resolution pictures of cracks on asphalt surfaces.

Alternatively, you may record a **short video** (30–60 seconds) of cracked surfaces and then extract clear, distinct frames to use as images. Free tools such as VLC or online frame extractors can help with this. If taking a video, avoid moving the camera too quickly to prevent motion blur.

Make sure **each image is unique**—avoid submitting multiple pictures of the same crack from similar angles. Your 50 images should look **visually different from each other**.

You need to upload your images to Quera, where the teaching assistants will review them.

💡 ***Hint**: Explore areas such as your home surroundings, daily commute route, or university campus to find cracks in sidewalks or roads. Parking lots are also good spots, as they often feature noticeable crack patterns.*

1.2. Label your dataset

After the teacher assistants have checked your images, it's time to label them using RoboFlow. RoboFlow is a powerful tool for annotating images. You can watch the TA's class on how to label your dataset for detailed guidance.

💡 ***Hint:** Consider using **Segment Anything (SAM)**, an image segmentation model developed by Meta Research that enables zero-shot segmentation.*

2. Preprocessing

After collecting and labeling your dataset, the preprocessing phase prepares your images for the training phase of the crack segmentation project. This step ensures your data is consistent, normalized, and ready for your chosen machine learning framework.

2.1. Share Your Project Link

Before diving into preprocessing, you need to share your work with the teaching assistants.

- **Task**: Upload the link of your Roboflow project (where you labeled your dataset) to the Quera site.
- **Purpose**: This allows the teaching assistants to review your labeled images of asphalt cracks and undamaged surfaces.

💡 ***Hint***: Double-check that your Roboflow project is public or accessible to the TAs before submitting the link.

2.2. Data Preprocessing

Now that your dataset is labeled, it's time to normalize your images to prepare them for training.

- **Objective**: Convert your images into a consistent format suitable for deep learning models.
- **Tools**:
    1. If using PyTorch: Leverage `DataLoader` and `Dataset` classes to manage and preprocess your data.
    2. If using Keras/TensorFlow: Use `ImageDataGenerator` or `tf.data.Dataset`, which offer similar functionality to PyTorch's `DataLoader` and `Dataset`.
- **Steps**:
    1. **Parse the COCO JSON File:**
        - Instead of manually reading the JSON file, the best practice is to use a specialized library designed for this format.
        - The `pycocotools` library (or `cocoapi`) is the official and most efficient tool for working with COCO datasets.
    2. **Convert Polygons to Pixel Masks:**
        - Your model needs a ground truth image where each pixel's value is either `0` (background) or `1` (crack).
        - For each image, create a blank, single-channel image (a NumPy array) of the same dimensions, filled with zeros. This will be your ground truth mask.
        - Use the `pycocotools` library to iterate through all the polygon annotations associated with that image.
        - For each polygon, use a function (like `coco.annToMask`) to rasterize the polygon—essentially "drawing" it onto your blank mask.
        - Assign the pixel values inside the drawn polygon to your target class ID (e.g., `1` for "crack"). If there are overlapping cracks, the later-drawn polygons will overwrite the earlier ones, which is acceptable for semantic segmentation.
    3. **Resizing:**
        - Standardize all images and masks to your model's required input size (e.g., 512×512 pixels).

    - **Important:** When resizing the pixel mask, always use **nearest-neighbor interpolation** to ensure the class labels remain discrete integers and don't get blended.
    - Scale your image's pixel values (e.g., divide by 255.0 to get values in the range).
4. **Normalization**:
    - Scale your image's pixel values (e.g., divide by 255.0 to get values in the range). **Important:** Do not normalize the pixel masks; their values should remain as integer class IDs.
5. **Build a Data Loader Pipeline:**
    - For PyTorch, apply `transforms.Normalize(mean, std)` after resizing.
    - For Keras/TensorFlow, use `ImageDataGenerator` or TensorFlow operations to normalize your data.

💡 ***Hint***: Check the documentation for your chosen model (e.g., ResNet or MobileNet) to confirm the expected input size.

2.3. Dataset Preparation for Training

With your images normalized, organize them for the training phase.

- **Task**: Split your dataset into training, validation, and test sets.
- **Guidelines**:
    - A common split is **70% training, 15% validation, and 15% test**.
    - For your 50 images (40 cracked, 10 undamaged), this could mean roughly 35 training, 7 validation, and 8 test images.
    - Ensure the split balances cracked and undamaged images to avoid bias.

2.4. Final Checks

Before moving to training, verify your preprocessing steps.

- **Tasks**:
    - Test your `DataLoader` (PyTorch) or `tf.data.Dataset`/`ImageDataGenerator` (Keras/TensorFlow) to ensure the dataset loads without errors.

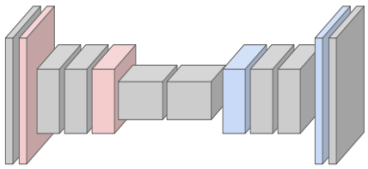
- Visualize a few processed images to check that cracks remain clear and undistorted.

3.Model implementation and training

With a well-prepared dataset, the next stage is to design, train, and evaluate our crack segmentation model. This section provides a detailed description of the model architecture, training procedure, and performance testing methodology.

3.1. Design the Model Architecture

If you want to design your own model from scratch, you will need to define the layers of a convolutional neural network (CNN). The general principle is to create an **encoder-decoder structure** to capture and then reconstruct image features at the pixel level. You can use standard layers like `Conv2D`, `MaxPooling2D`, `UpSampling2D`, and activation functions like `ReLU` to build your network. You'll likely want to include skip connections to pass fine-grained information from the encoder to the decoder.



🚨 **Implementation:** We will not use pre-designed architectures (like U-Net) or pre-trained weights. Furthermore, in this section, you should not use any dropout layers or other explicit regularization techniques. The emphasis is on good model training on the dataset and the original architecture design.

This entire architecture will be implemented using PyTorch or TensorFlow Keras layers in the model.py file.

3.2. Train the Model

The training process involves feeding our dataset to the model and optimizing its weights using a loss function. The implementation will be contained in train.py.

🚨 **Implementation:** Before training begins, our dataset is split into two primary subsets: a **training set** and a **validation set**. The model learns directly from the training set, while the validation set is used to evaluate its performance on unseen data at the end of each epoch. This process is crucial for monitoring the model's ability to generalize.

To monitor the learning process, we will plot the loss on both the training and validation sets against the number of epochs. Analyzing this graph is essential for diagnosing the model's behavior. Results and plots are required for next steps.

🧩 **Option:** To handle the large dataset efficiently, especially on systems with limited RAM, we will implement a **custom data generator** (in Keras) or **PyTorch Dataset class**. This approach loads and processes only a small batch of images at a time, making the training process feasible even on low-resource computers.

The key components of our training loop are:

- **Loss Function**
- **Optimizer**
- **Hyperparameters**

A standard loss function like Binary Cross-Entropy is often insufficient for crack segmentation due to the extreme class imbalance (cracks are tiny compared to the background). Do you have any idea?

💡 **Hint:** It is highly recommended to use cloud-based environments such as Google Colab for training. These platforms provide free access to GPUs, which can significantly speed up the training process compared to a standard CPU.

3.3. Evaluate the Model

Now that your model is trained, it's time to assess its performance on the test dataset. In this part, you will implement the evaluation step in the evaluate.py file by focusing on a single, widely-used metric in segmentation tasks: Intersection over Union (IoU).

$$\text{IoU} = \frac{|\text{Prediction} \cap \text{Ground Truth}|}{|\text{Prediction} \cup \text{Ground Truth}|}$$

IoU quantifies the overlap between the predicted mask and the ground truth by dividing the area of their intersection by the area of their union. To perform the evaluation, complete the evaluate.py file by loading the trained model in evaluation mode and generating predicted crack masks for the test images. For each test image, compute the IoU score between the predicted binary mask and its corresponding ground truth mask, and then calculate the average IoU across the entire test dataset. Make sure to threshold the model's output to obtain a binary mask before computing the IoU. Finally, visualize a few results by overlaying the predicted mask on the original image to qualitatively assess your model's performance. This evaluation will help you understand how accurately your model detects cracks and how well it generalizes to new data.

3.4 Pipeline the process

You should create a pipeline to automate the inference process. This function will take a raw image as input and perform the following steps:

1. **Preprocess the image**
2. **Predict the mask:** Pass the preprocessed image through the trained model to obtain a segmentation mask.
3. **Post-process the output:** Resize the predicted mask back to the original image's dimensions to allow for direct comparison and visualization.
4. **visualize the model's output**

4. Enhance the Dataset with Augmentation

Data augmentation artificially increases the variety and size of the training dataset, helping the model learn to detect cracks under diverse conditions and reducing overfitting. In crack segmentation, cracks can appear at various angles, lighting conditions, and backgrounds.

4.1 Choosing Transformations

For crack segmentation, we have selected the following transformations based on their relevance to the task:

- **Rotation:** Cracks can appear at various angles on asphalt surfaces. Rotating the images helps the model learn to detect cracks regardless of their orientation. We will apply random rotations between -30 and 30 degrees.
- **Scaling**: Cracks can vary in size depending on their distance from the camera. Scaling the images helps the model detect cracks at different scales. We will apply random scaling between 0.8 and 1.2 times the original size.
- **Brightness and Contrast Adjustments**: Lighting conditions can differ due to weather or time of day. Adjusting brightness and contrast helps the model detect cracks under diverse lighting scenarios. We will apply random brightness adjustments between -20% and 20% and random contrast adjustments between -20% and 20%.
- **Noise Addition**: Real-world images often contain noise from camera sensors or environmental factors. Adding noise helps the model learn to detect cracks in noisy conditions. We will add Gaussian noise with a standard deviation of 10. We will avoid using flipping or cropping. Flipping may not be necessary since cracks are not inherently symmetric, and cropping could remove critical parts of the cracks, hindering the model's learning process.

4.2. Applying Transformations

When applying transformations, it is essential to ensure that the same transformations are applied to both the images and their corresponding segmentation masks. This maintains the accuracy of the labels after augmentation. For example, if an image is rotated by 30 degrees, its segmentation mask must also be rotated by 30 degrees to align the crack annotations correctly.

4.3. Augmentation Pipeline

The augmentation pipeline consists of the following transformations, applied in sequence:

1. Random rotation between -30 and 30 degrees.
2. Random scaling between 0.8 and 1.2 times the original size.
3. Random brightness adjustment between -20% and 20%.
4. Random contrast adjustment between -20% and 20%.
5. Addition of Gaussian noise with a standard deviation of 10.

For each original image, we will generate 5 augmented versions using this pipeline. This will increase the dataset size by a factor of 5, enhancing the diversity of training data.

4.5. Verification

To verify that the augmentations are correctly applied, visually inspect a sample of augmented images and their segmentation masks. Ensure that the transformations are consistent between the images and masks—for instance, check that a rotated image has a correspondingly rotated mask. Additionally, review the augmented dataset to confirm it reflects a wide range of variations in orientation, scale, and lighting. Be cautious of excessive augmentation, which might distort cracks or introduce artifacts that confuse the model. Since augmentation increases the dataset size, ensure you have sufficient computational resources and expect longer training times.

💡 **Hint:** Before applying augmentation, split your dataset into training, validation, and test sets. Apply augmentation only to the training set to ensure the model is evaluated on original, unaugmented data.

🚨 **Implementation note:** You are not allowed to use any third-party libraries to handle transformations. You **MUST** use the techniques learned in the course and *OpenCV functions* to apply augmentation.

5. Regularize Your Model Using Dropout

In this part, you will improve your model's generalization capability by adding Dropout layers. Dropout is a simple yet powerful regularization technique that helps prevent overfitting, especially when working with small or imbalanced datasets like crack segmentation, where crack pixels are often rare compared to background pixels. During training, Dropout randomly sets a fraction of neuron outputs to zero with a certain probability (e.g., 0.2 or 0.5). This prevents the model from becoming too reliant on specific neurons and encourages it to learn more robust and distributed features. Now, update your segmentation model by inserting Dropout layers and train it again to observe the

difference in performance compared to the model without Dropout, and report the impact on accuracy and generalization. What are other ways to make our model robust on unseen data?

6. Optional: Transfer learning from Pre-trained Models

While building a model from scratch provides invaluable foundational knowledge, leveraging models that have been pre-trained on vast datasets can often yield superior performance with less training time.

- **How it works:** You use the pre-trained weights for the encoder part of your segmentation model (the part that learns to extract features from the image). The encoder has already learned to recognize fundamental patterns like edges, textures, and shapes.
- **Applying it:** You then only need to train the decoder part of the network, which learns how to reconstruct the segmentation mask specifically for cracks.
- **Benefit:** The model starts with a strong foundation of learned features, which allows it to converge faster and achieve better performance with less data, as it doesn't have to learn these basic features from scratch. Common choices for pre-trained encoders include `ResNet`, U-Net, `BiseNet`, and `MobileNetV3`

Compare the performance of the fine-tuned model against the custom-built CNN from Part 3 to quantify the benefits of transfer learning.

K. N. Toosi
University of Technology

Submission Criteria

To ensure a successful evaluation of your crack segmentation project, the following criteria must be met in your submission:

- **Code Implementation & Explanation**
  - Provide a well-documented, modular, and executable codebase.
  - Demonstrate a strong grasp of all concepts (e.g., preprocessing, model architecture, loss functions, post-processing).
  - Be prepared to explain and modify any part of the code if asked (e.g., adjusting hyperparameters, changing the model architecture).
  -
- **Visualization of Results**
  - Visualize intermediate and final outputs (e.g., input image, ground truth mask, predicted segmentation, edge detection if applicable).
  - Include comparisons between different methods (if experimented) with qualitative analysis.
- **Pipeline for Unseen Data**
  - Develop a complete inference pipeline that can:
  - Accept an unseen crack image as input.
  - Preprocess it appropriately (resizing, normalization, etc.).
  - Generate a segmentation mask with clear crack labels.
  - Ensure the pipeline is robust to variations (e.g., lighting, noise, different surfaces).
- **Performance Metrics & Analysis**
  - Report quantitative metrics (e.g., IoU, Dice score, precision/recall) on test data.
  - Discuss limitations and potential improvements (e.g., handling thin cracks, false positives/negatives).

📌**Note**: Submissions that fail to meet these criteria may be returned for revisions. Focus on clarity, functionality, and demonstrating deep understanding.

References & Resources

- https://roboflow.com/model/segment-anything-model-sam