Classification, Segmentation, and GAN Implementation on Concrete Crack Images

Research and Development project

Hrithik Mhatre and Vamshika Sutar

Under the Supervision of

Prof. Abir De and Prof. Alankar Alankar

Department of Civil Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

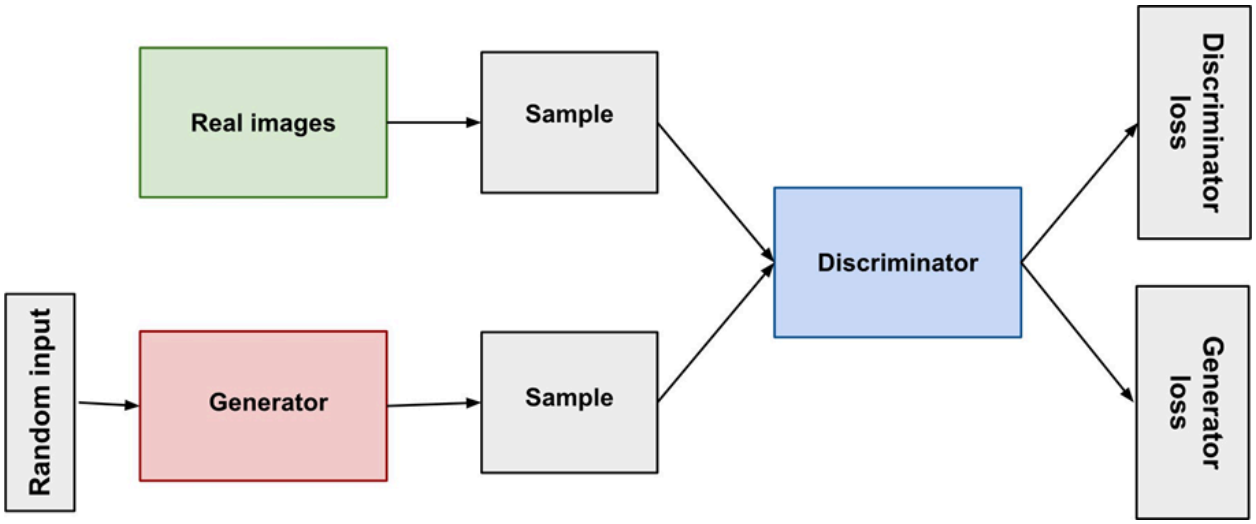
Mumbai - 400076,

India

1. Introduction

Early detection of surface cracks is essential for ensuring the safety and longevity of civil infrastructure. Traditional manual inspections are subjective and error-prone, whereas deep learning offers a scalable and consistent alternative. However, real-world datasets often suffer from class imbalance, with far fewer cracked images, leading to biased models. To address this, we propose an integrated framework combining Generative Adversarial Networks (GANs) for minority class augmentation, a ResNet-based classifier for crack detection, and a U-Net model for pixel-level crack segmentation. GANs generate realistic cracked surface images, mitigating data imbalance, while the ResNet and U-Net architectures provide robust classification and precise localization, respectively. This pipeline offers a comprehensive solution for automated structural health monitoring, particularly in data-scarce environments.

2. Generative Adversarial Networks - GAN



2.1 Introduction

In computer vision tasks, the performance of deep learning models is heavily reliant on the availability of large and diverse datasets. However, in many practical scenarios, especially in domains such as structural health monitoring and defect detection, the availability of labeled training data is severely limited. In our case, the dataset exhibited significant data scarcity, impeding the development of a robust image classification model for crack detection.

To overcome this challenge, we employed Generative Adversarial Networks (GANs) to augment the dataset by synthesizing realistic cracked surface images, thereby enhancing the training

distribution. This section outlines the architecture, training methodology, and implementation details of the GAN framework used for this purpose.

2.2 Generative Adversarial Networks: Framework Overview

A Generative Adversarial Network (GAN) is a class of deep generative models comprising two neural networks, the Generator and the Discriminator, which are trained in an adversarial setup. The Generator (G) attempts to produce realistic synthetic images from random noise vectors. The Discriminator (D) attempts to distinguish real images from those generated by G. This adversarial process drives both networks to improve simultaneously: the Generator learns to produce increasingly realistic images, while the Discriminator becomes more adept at classification.

GAN Architecture Overview

Generator: A deep convolutional neural network that performs upsampling using transposed convolutional layers.

Discriminator: A convolutional neural network that performs downsampling and outputs a scalar probability indicating whether the input is real or generated.

2.3 Model Hyperparameters

The GAN was trained with the following configuration:

1. *Number of workers: 2*
2. *Batch size: 64*
3. *Image spatial size: 64 × 64*
4. *Number of channels (nc): 3*
5. *Latent vector size (nz): 100*
6. *Generator feature map size: 64*
7. *Discriminator feature map size: 64*
8. *Number of epochs: 60*
9. *Learning rate: 0.0002*
10. *Adam optimizer β_1: 0.5*
11. *Number of GPUs: 1*

2.4 Image Preprocessing

Prior to training, all input images (both real and synthetic) underwent the following preprocessing steps to ensure uniformity and compatibility with PyTorch:

1. Resized to 64×64 pixels.
2. Center-cropped for spatial consistency.
3. Converted to tensor format.
4. Normalized using a mean of (0.5, 0.5, 0.5) and standard deviation of (0.5, 0.5, 0.5) to standardize pixel values in the range [–1, 1].

2.5 Data Loading

To facilitate efficient training, the torch.utils.data.DataLoader module was employed with parameters:

1. *Batch size: 64*
2. *Shuffle: True*
3. *Number of workers: 2*

2.6 Weight Initialization

Weight initialization followed the standard GAN procedure:

Convolutional Layers: Initialized with a normal distribution (mean = 0, std = 0.02).

Batch Normalization Layers: Weights set to 1, biases to 0, initialized from a normal distribution.

2.7 Generator Architecture

The Generator transforms latent noise vectors into high-fidelity synthetic images.

Input: A noise vector of size nz = 100

Layers:

Five ConvTranspose2d layers increase spatial dimensions and reduce feature map depth

BatchNorm2d and ReLU applied after each upsampling layer

Final Tanh activation to constrain pixel values to [–1, 1]

Output: RGB image of dimensions (3 × 64 × 64)

2.8 Discriminator Architecture

The Discriminator classifies images as real or synthetic.

Input: RGB image of dimensions ($3 \times 64 \times 64$)

Layers:

Five Conv2d layers downsampling the image and increasing feature depth

BatchNorm2d and LeakyReLU activations to stabilize gradients

Final Sigmoid activation to produce a scalar output in [0, 1]

Output: A single probability value representing image authenticity

2.9 Optimizer and Training Strategy

Both Generator and Discriminator networks were trained using the Adam optimizer.

1. *Learning rate: 0.0002*
2. β_1*: 0.5*
3. β_2*: default (0.999)*

The training was conducted over 60 epochs with alternating updates to the Generator and Discriminator.

2.10 Loss Functions

Discriminator Loss:

$L_D = -[\log D(x) + \log(1 - D(G(z)))]$

Generator Loss:

$L_G = -\log(D(G(z)))$

2.11 Training Loop

Step 1: Train Discriminator

Real images: Compute real loss

Fake images: Compute fake loss

Total loss: Update D’s weights

Step 2: Train Generator

Generate images from noise

Compute loss based on D’s classification

Update G to improve realism

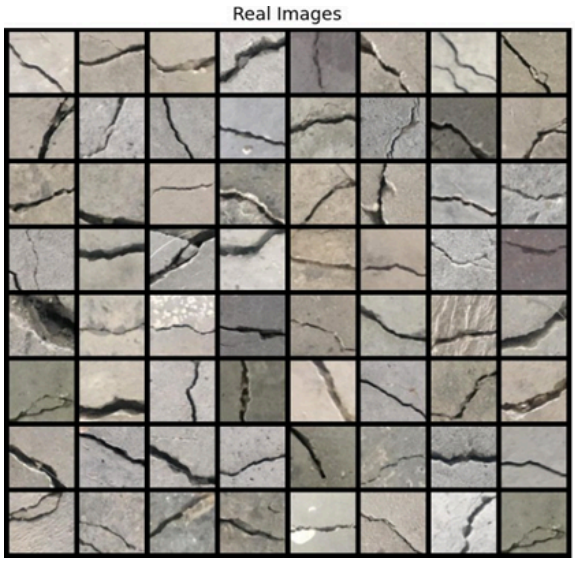
Step 3: Logging

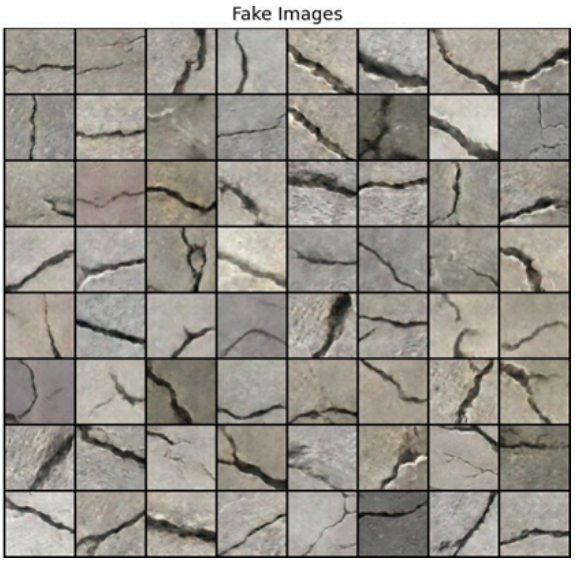
Losses, D(x), D(G(z)) tracked per epoch

Generated samples stored for visual inspection

2.12 Outcome

The application of GANs enabled the generation of high-quality synthetic images, effectively mitigating the effects of data scarcity. These images enriched the training set and improved the robustness of downstream classification models, validating the efficacy of the adversarial training framework.





Concrete Crack Detection Utilizing ResNet50

3.1 Dataset

The dataset used in this study consisted of a limited number of labeled images of cracked and non-cracked concrete, each with a resolution of 227 × 227 pixels and formatted in RGB JPEG.

To overcome data scarcity and enhance the diversity of training samples, we employed a Generative Adversarial Network (GAN) to synthesize realistic cracked concrete surface images. These GAN-generated images were combined with the original dataset to form an expanded training set, to improve robustness and generalisation.

3.2 Data Generator and Image Preprocessing

1. **Data Augmentation**: An ImageDataGenerator (train_datagen) was configured with augmentation techniques to artificially expand the dataset and mitigate overfitting. This step enhances model generalization.

2. **Preprocessing**: The preprocess_input function from tensorflow.keras.applications.ResNet50 standardizes the input by converting RGB to BGR and centering pixel values relative to the ImageNet dataset.

3. **Validation Split**: A 20% validation split was specified during training data generation to monitor model performance during training.

4. **Testing Generator**: A separate ImageDataGenerator (test_datagen) without augmentation was used for the test dataset to maintain evaluation integrity.

5. **Flow from DataFrame**: Data generators (train_gen, valid_gen, and test_gen) were created using the flow_from_dataframe method, which loads images based on file paths and labels stored in a DataFrame.

6. **Image Configuration**: All images were resized to (100 × 100) pixels. RGB mode was maintained throughout to ensure consistency.

7. **Class Mode**: Set to 'categorical' for binary classification across two classes.

8. **Batch Size and Shuffling**: A batch size of 64 was used. Shuffling was enabled for training (shuffle=True) and disabled for validation and testing (shuffle=False).

9. **Return**: The function returned the three generators: train_gen, valid_gen, and test_gen for training, validation, and testing, respectively.

3.3 Model Creation

1. **Model Initialization**: A pre-trained **ResNet50** model was initialized with input shape (100, 100, 3).

2. **Pre-trained Weights**: Weights from the ImageNet dataset were used (weights='imagenet') to leverage existing visual feature representations.

3. **Feature Extraction**: The model excluded the top classification layers (include_top=False) to allow for custom layers suited to our specific task.

4. **Freezing Layers**: The base ResNet50 layers were frozen (pre_model.trainable = False) to retain pre-trained features and reduce training time.

5. **Custom Classification Head**: Two Dense layers with ReLU activation were appended, followed by a Softmax layer with two output units for binary classification.

6. **Compilation**: The model was compiled with:

- **Loss**: categorical_crossentropy
- **Optimizer**: Adam
- **Metric**: Accuracy

7. **Early Stopping**: An EarlyStopping callback monitored validation loss, halting training if no improvement was seen for one epoch (patience=1).

8. **Return**: The function returned the compiled model and my_callbacks for training use.

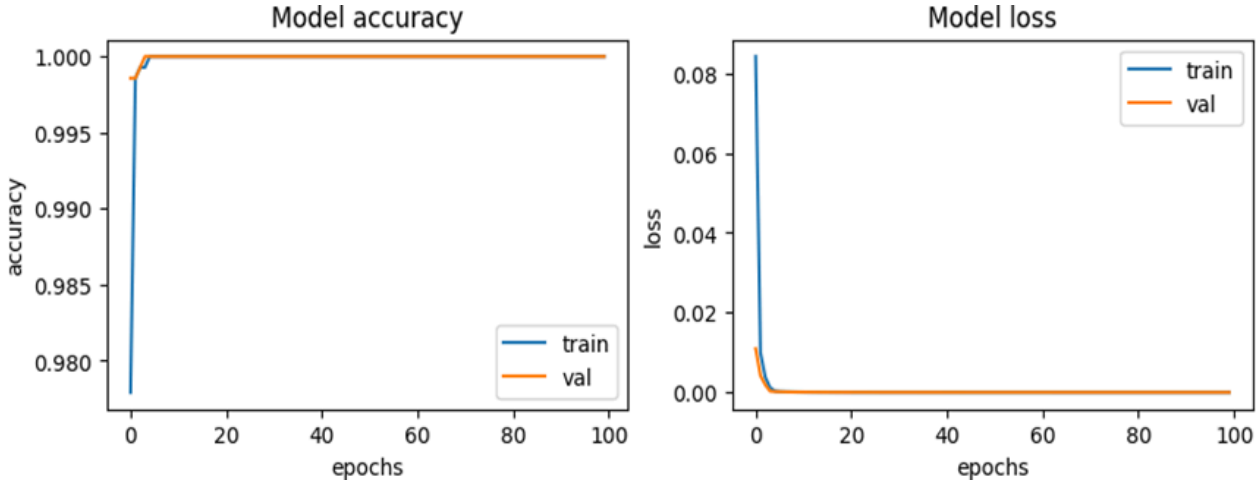
3.4 Training the Model

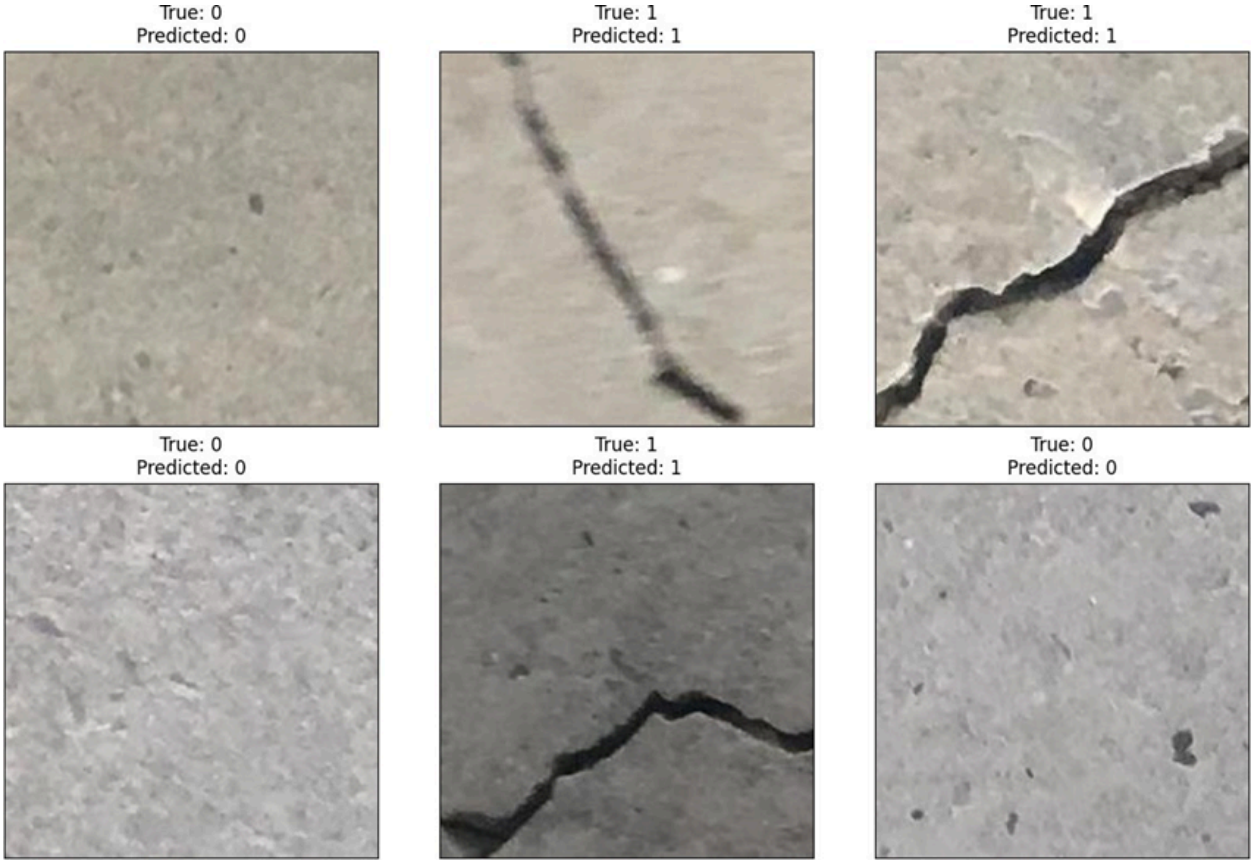
The model was trained for up to **100 epochs** using the model.fit() method with the previously defined early stopping criteria. The training utilized both the original and GAN-generated images to improve robustness and generalization performance.

3.5 Results

3.5.1 Performance Evaluation on the Training Dataset

	precision	recall	f1-score	support
0	0.99	1.00	1.00	295
1	1.00	0.99	1.00	307
accuracy			1.00	602
macro avg	1.00	1.00	1.00	602
weighted avg	1.00	1.00	1.00	602


Model accuracy
accuracy
epochs
train
val
Model loss
loss
epochs
train
val


True: 0
Predicted: 0
True: 1
Predicted: 1
True: 1
Predicted: 1
True: 0
Predicted: 0
True: 1
Predicted: 1
True: 0
Predicted: 0

3.5.2 Evaluation on External Datasets Using Pre-Trained Weights

To evaluate generalization capability, the trained model was tested on two external datasets:

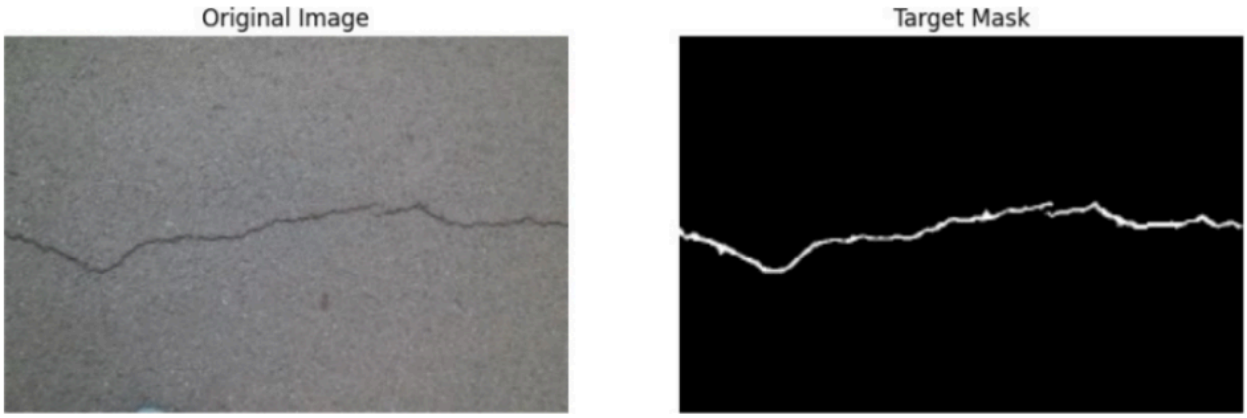
- **Dataset**: hesighsrikar/concrete-crack-images-for-classification
    - **Test Accuracy**: 99.33%
    - **F1 Score**: 0.99
    - **ROC AUC**: 0.99
- **Dataset**: aniruddhsharma/structural-defects-network-concrete-crack-images
    - **Test Accuracy**: 82.02%
    - **F1 Score**: 0.77
    - **ROC AUC**: 0.59

The reduced performance on the second dataset can be attributed to the presence of mislabeled or ambiguous images. Nonetheless, the model retained a reasonable level of predictive power, demonstrating strong generalization from the combined training set of original and GAN-generated images.

4. Crack Segmentation using U-Net

4.1 Dataset

Cracked Concrete surface images generated from the GAN and original dataset were used for the segmentation task. We have annotated a crack image database. We have used 80% data for the training set and 20% for the testing. Example image from the dataset:



4.2 Pre-Processing

4.2.1 **Resizing**

We resized the images to be of size 240x160 to improve computational efficiency. In this size, they retain sufficient information and are computationally feasible.

4.22 **Normalizing**

Divided all pixel values by 255 so they are in the range 0 to 1.

4.2.3 **Additional preprocessing on Mask:**

Extracted the alpha channel from the PNG masks and converted the image to Grayscale after binary thresholding the pixels.

4.3 U-net Architecture

U-Net is a widely used deep learning architecture that was first introduced in the paper titled

“U-Net: Convolutional Networks for Biomedical Image Segmentation” by Olaf Ronneberger et al. in 2015 at the University of Freiburg, Germany. Its primary purpose was to address the challenge of limited annotated data in the medical field.

Key Characteristics of U-Net:

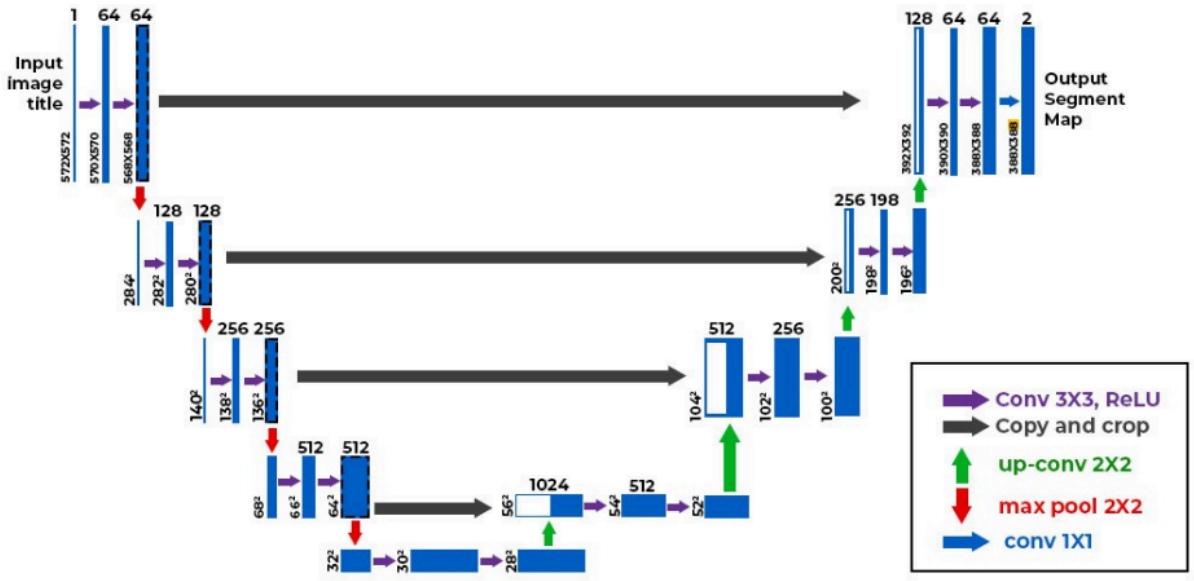
1. **Unique U-Shaped Structure:**
    - The U-Net architecture derives its name from its distinctive U-shaped structure.
    - It consists of two main parts: the **contracting path (encoder)** and the

expanding path (decoder).

2. **Contracting Path (Encoder):**
    - The contracting path captures contextual information and reduces the spatial resolution of the input.
    - Encoder layers perform convolutional operations, progressively reducing the spatial resolution of feature maps while increasing their depth.
    - These layers capture increasingly abstract representations of the input, similar to feedforward layers in other convolutional neural networks.
3. **Expanding Path (Decoder):**
    - The expanding path decodes the encoded data and locates features while maintaining the spatial resolution of the input.
    - Decoder layers upsample the feature maps while also performing convolutional operations.
    - **Skip connections** from the contracting path help preserve spatial information lost during the contraction, aiding the decoder layers in locating features accurately.
4. **Segmentation Map Generation:**
    - U-Net generates a segmentation map (output) based on the input image.
    - Each pixel in the output map represents a label corresponding to a particular object or class in the input image.
    - In biomedical applications, this map can represent foreground and background regions, aiding in tasks like tumor segmentation.

Illustrating U-Net:

- Figure 12 illustrates how the U-Net network converts a grayscale input image of size 572×572×1 into a binary segmented output map of size 388×388×2.
- Notice that the output size is smaller than the input size because no padding is used. However, using padding can maintain the input size.



Contracting Path:

- The input image progressively reduces in height and width but increases in the number of channels (feature maps).
- This increase in channels allows the network to capture high-level features as it progresses down the path.

Expanding Path:

- The feature map from the bottleneck is converted back into an image of the same size as the original input.
- Upsampling layers increase the spatial resolution of the feature map while reducing the number of channels.
- Skip connections from the contracting path help the decoder layers locate and refine features in the image.

4.4 Model

4.4.1 Libraries used:

- TensorFlow
- OpenCV
- NumPy
- Pandas
- Sci-kit learn

4.4.2 Hyperparameters:

- Batch size = 8

- Learning rate = 10e-3
- Training size = 80%
- Number of epochs = 100
- Patience = 30 (Dice coefficient monitored)

4.4.3 Evaluation

- Loss function: Dice coefficient loss
- Optimizer: Adam
- Performance measure: Dice coefficient, Mean IoU

4.4.4 Results (On training set)

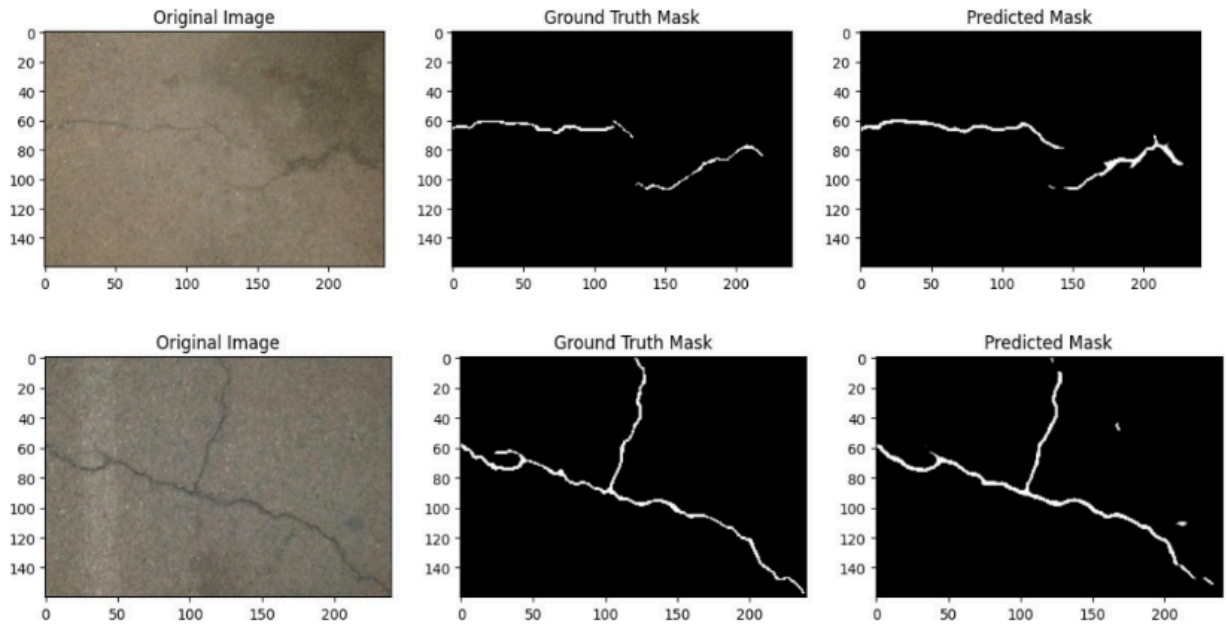
```
Epoch 95/100
11/11 [==============================] - 2s 226ms/step - loss: 0.2493 - dice_coef: 0.7501 - mean_io_u_2: 0.6190
Epoch 96/100
11/11 [==============================] - 2s 226ms/step - loss: 0.2512 - dice_coef: 0.7503 - mean_io_u_2: 0.5982
Epoch 97/100
11/11 [==============================] - 2s 223ms/step - loss: 0.2500 - dice_coef: 0.7503 - mean_io_u_2: 0.6193
Epoch 98/100
11/11 [==============================] - 2s 224ms/step - loss: 0.2671 - dice_coef: 0.7323 - mean_io_u_2: 0.6174
Epoch 99/100
11/11 [==============================] - 2s 223ms/step - loss: 0.2630 - dice_coef: 0.7366 - mean_io_u_2: 0.6048
Epoch 100/100
11/11 [==============================] - 2s 226ms/step - loss: 0.2472 - dice_coef: 0.7526 - mean_io_u_2: 0.6125
1/1 [==============================] - 0s 169ms/step
```

4.4.5 Results (On test dataset)

```
142/Unknown - 11s 53ms/step - loss: 0.2856 - dice_coef: 0.6977 - mean_io_u_2: 0.6185
```

4.4.6 Visualizing results

Example results from the model's predictions on test dataset are shown in figure 13 below



5. Conclusion

In conclusion, our research and development project delved into the crucial domain of concrete crack detection and analysis. Through the implementation of classification and segmentation techniques, we aimed to enhance the efficiency and accuracy of crack identification in concrete structures.

In the realm of classification, we explored various datasets and models, leveraging machine learning algorithms such as ResNet50. Through rigorous evaluation on diverse datasets, our models demonstrated robust performance, achieving high F1 scores and ROC scores.

Moving to segmentation, we employed the U-Net architecture. Our model exhibited promising results on the test set , accurately delineating cracks in road images.

In essence, our endeavor contributes to advancing the field of concrete crack detection, emphasizing the significance of automated methods in enhancing structural integrity, safety, and maintenance practices. However, further exploration and refinement are warranted, particularly in addressing the nuances of crack segmentation in diverse environmental and structural contexts.