Concrete Crack Detection and Segmentation

Authors

- **Yu-Sian Lin, Chengyou Yue, Yueh-Ti Lee, Minjiang Zhu**

Department of Civil and Environmental Engineering, University of Illinois

Introduction

The existence of cracks in concrete materials is of vital importance in Civil Engineering. An accurate and fast method to detect the existence of cracks in concrete images is always sought by engineers. Nowadays, even though a Convolutional neural network (CNN) based model can easily tell the existence of concrete cracks, the information on the exact locations of cracks is smeared in the network. We aim first to build a CNN-based model for crack detection and then apply another model called U-Net to locate and show the exact positions of cracks. We also solved the problem that U-Net would recognize dark areas as cracks and an overfitting problem.

We first train a CNN with the Surface Crack Detection dataset [1] to judge if any concrete crack exists. The dataset contains images (each has 227×227 pixels and RGB channels) of concrete surfaces with or without cracks. Each class contains 20000 images. Examples are shown in Figure 1:

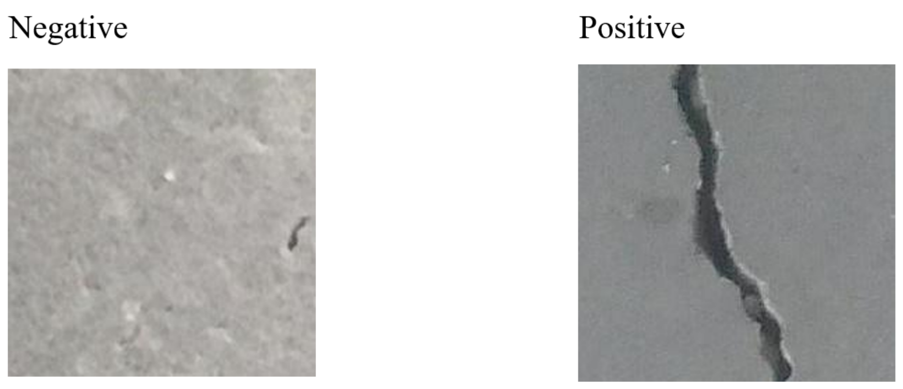


Figure 1: Negative and Positive Examples from the Surface Crack Detection Dataset

The dataset is split into a training set with 4200 images and a testing set with 1800 images randomly selected from the dataset.

We then train an FCN-based network called U-Net [2] with the Concrete Crack Conglomerate Dataset [3] to segment the concrete from its cracks.

The dataset contains over 20000 crack images which include subsets named CFD, CRACK500, CRACKTREE etc. (each has 448×448 pixels and RGB channels). Each image data has an original image and a mask image. Examples are shown in Figure 2:

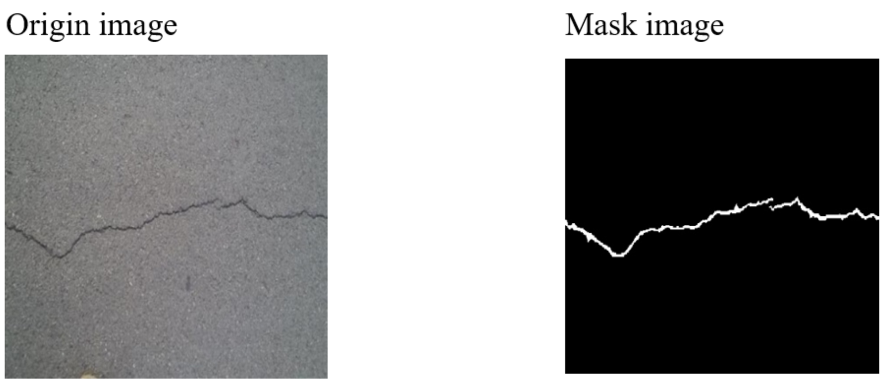


Figure 2: Origin and Mask Images from the Concrete Crack Conglomerate Dataset

U-Net is an elegant architecture that can work with very few training images and yield precise segmentations. [2] Different from the large number of images used in classification, we only chose the CFD sets with 107 images for training and 11 for testing.

Crack Detection by CNN

A Brief Introduction to CNN

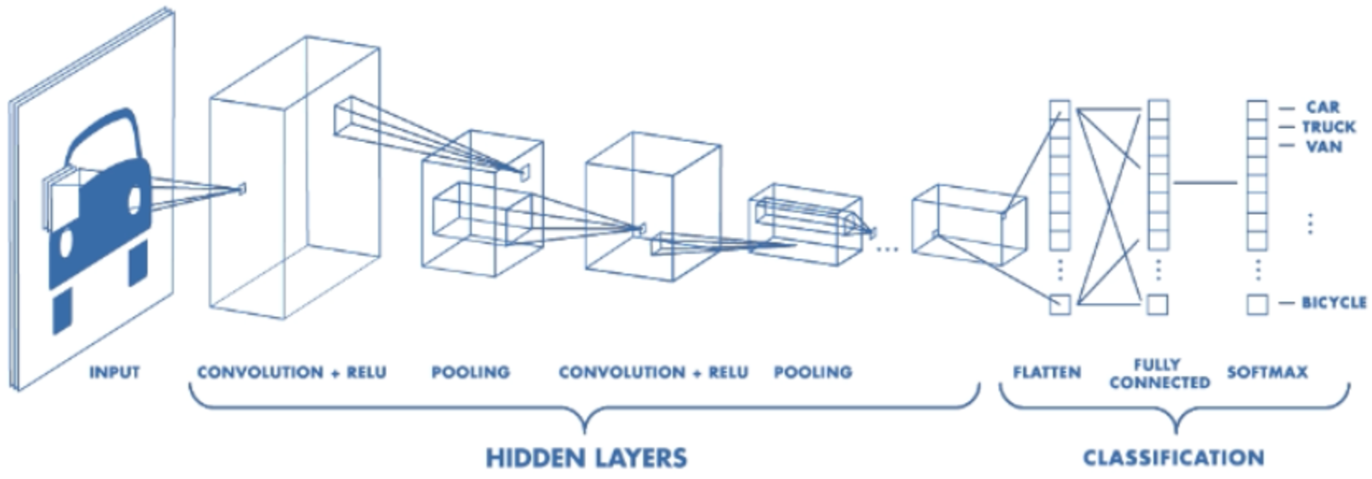
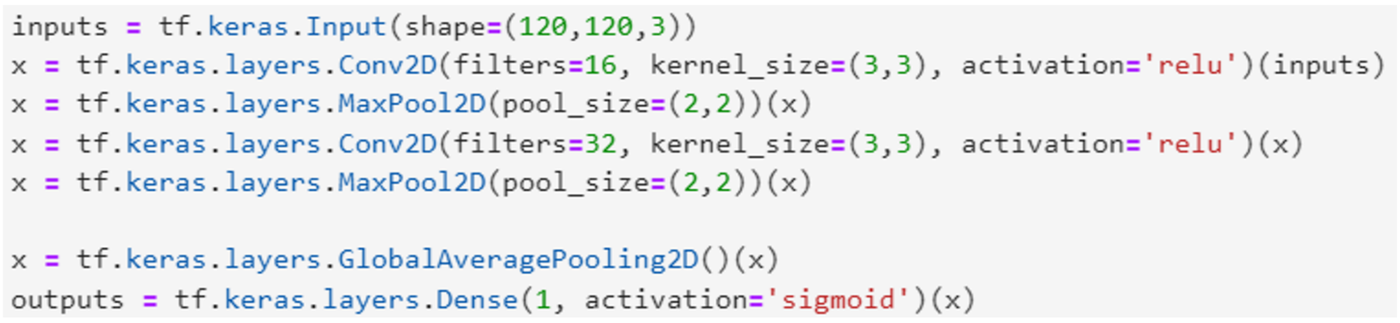


Figure 3: Basic Convolutional Neural Network for Classification.

CNN network is briefly introduced in this section. As shown in Figure 3, a typical CNN network involves several convolutional and pooling layers and several fully connected layers. Convolution is an operation that uses filters to extract information that we want to detect. The filter size determines how many filters will be applied to the input data. The kernel size determines the size of the area to which the filters would apply. Activation functions are added to import nonlinearity into the model, considering the linear filter operation. MaxPooling is a down-sampling operation that allows our network to capture deeper information from its original dimensions. The pool size and strides determine the dimensions of the down-sample procedure. Notice that the sigmoid function (rather than the softmax function shown in Figure 3) will be applied in our model, as we only have two classes of results (True or False).

Network Design

In the first convolutional block, we would specify 16 filters (consider 8 straight lines and 8 curves) with 3 by 3 kernels (considering the size of the crack is relatively small), assign ReLu as the activation function in the Conv2D layer, and go deeper with a pool size of 2 by 2 in the MaxPool2D layer. In the second convolutional block, we double the channel number to 32 with Conv2D and apply the same MaxPool2D. Finally, we apply a GlobalAveragePooling2D layer, focusing on the whole image rather than a part of it in segmentation. The coding form of this model is shown in Figure 4:



```
inputs = tf.keras.Input(shape=(120,120,3))
x = tf.keras.layers.Conv2D(filters=16, kernel_size=(3,3), activation='relu')(inputs)
x = tf.keras.layers.MaxPool2D(pool_size=(2,2))(x)
x = tf.keras.layers.Conv2D(filters=32, kernel_size=(3,3), activation='relu')(x)
x = tf.keras.layers.MaxPool2D(pool_size=(2,2))(x)

x = tf.keras.layers.GlobalAveragePooling2D()(x)
outputs = tf.keras.layers.Dense(1, activation='sigmoid')(x)
```

Figure 4: Model for Concrete Crack Classification using_Keras

The result of our network would be a number in (0,1). We may treat it as the probability of the existence of cracks in the image. Noticing that the original images have the size (227,227,3), we would resize it into (120,120,3) in the preprocessing procedure. The hyperparameters are shown in Table 1:

Table 1: Hyperparameters

Training	Validation frac in training	Testing	Optimizer	Loss Function	Metrics	Max Epochs
4200	20%	1800	Adam	Binary Crossentropy	Accuracy	100

Results and Analysis

Below is the confusion matrix our model generated:

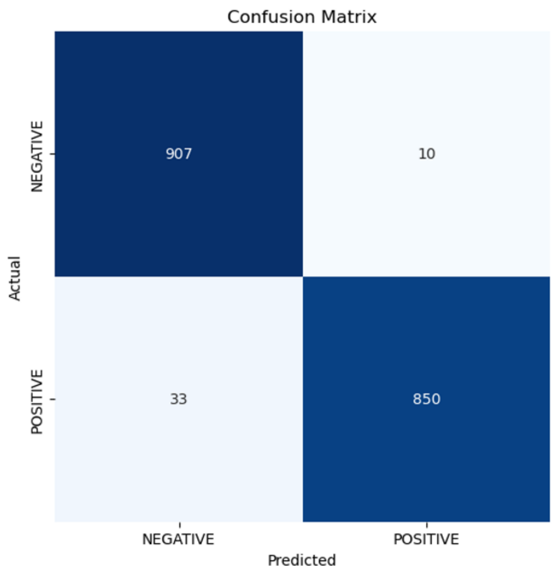


Figure 5: Confusion Matrix

and the loss during the training process:

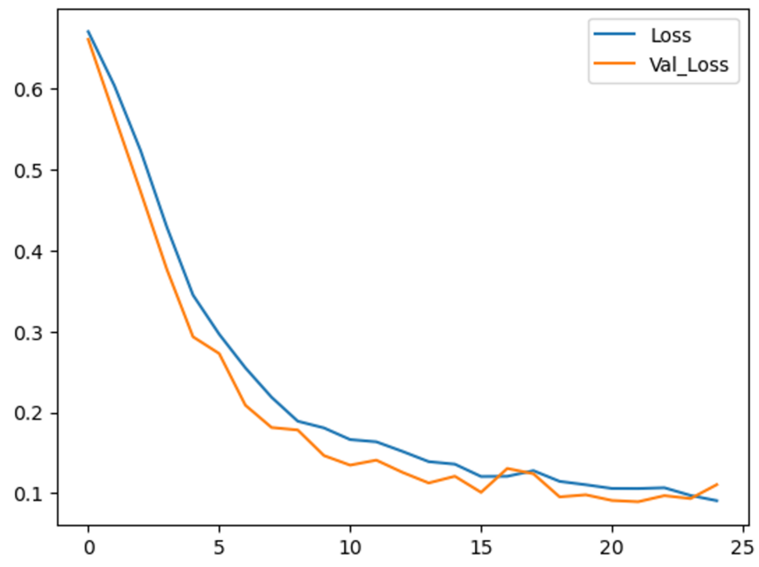


Figure 6: Loss Evolution

We can see that this trained model performs well on the testing data. The training process only lasts for 25 epochs because of the early-stop procedure. We can further calculate the following parameters:

$$\begin{aligned}\text{Precision} &= \frac{TP}{TP+FP}\\ \text{Recall} &= \frac{TP}{TP+FN}\\ \text{Score} &= 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}\\ \text{MacroAvg} &= \frac{1}{n}\sum_{i=1}^{n}\text{Score}_i\\ \text{WeightedAvg} &= \frac{1}{n}\sum_{i=1}^{n}\text{Support}_i \times \text{Score}_i\end{aligned} \tag{1}$$

Their specific values are shown in Table 2:

Table 2: Result Analysis

	Precision	**Recall**	**Score**	**Support**
Negative	0.96	0.99	0.98	917
Positive	0.99	0.96	0.97	883
Macro Avg	0.98	0.97	0.97	1800
Weighted Avg	0.97	0.97	0.97	1000

In short, the network has overall good performance.

Crack Segmentation by U-Net

A Brief Introduction to U-Net

U-Net is built upon the so-called “Fully Convolutional Network”, it was first introduced in 2015 and won the ISBI challenge for segmentation of neuronal structures in electron microscopic stacks. The original U-Net architecture is shown below:

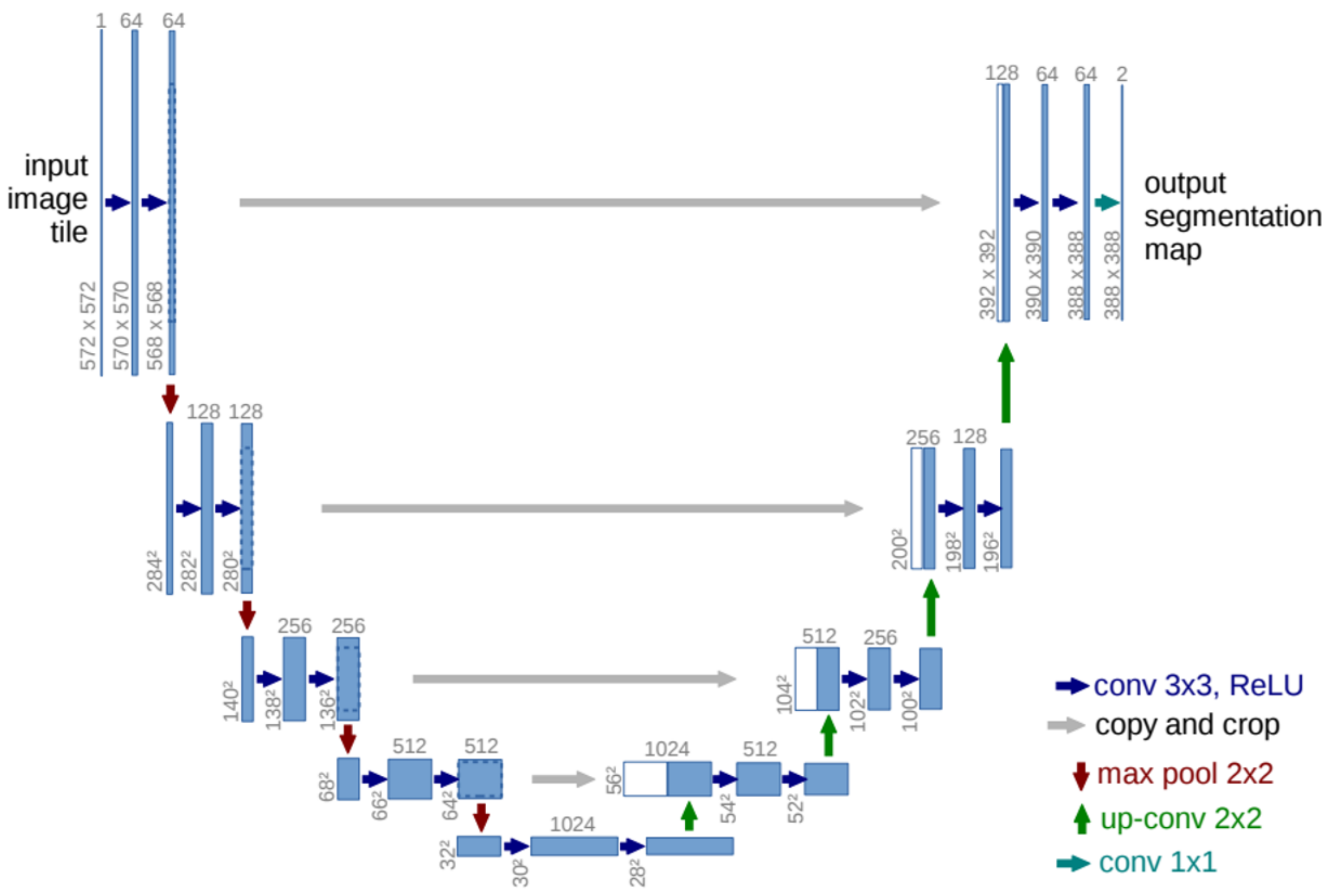


Figure 7: U-net Architecture.

Different from FCN, it supplements a usual contracting network with successive layers, where pooling operators are replaced by upsampling operators. It would keep the output from each convolutional layer by either concatenating it with the unsampled results or simply adding them together. This modification can take advantage of these procedures to overcome the trade-off between localization accuracy and the use of context (FCN requires more max-pooling layers which reduce the localization accuracy, while small patches allow the network to see only little context).

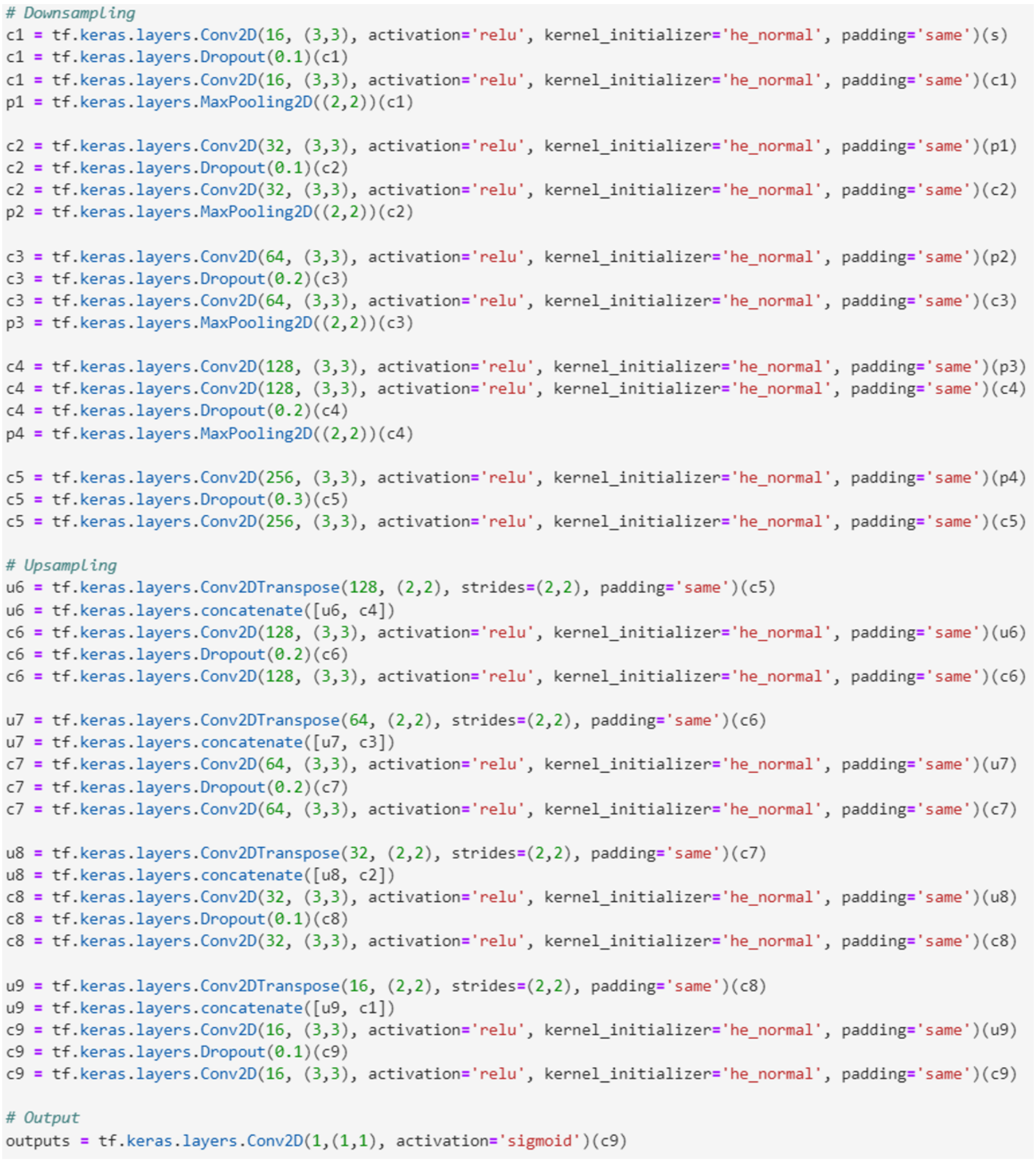
Modeling and Training

To achieve segmentation, we have built and trained two models, one based on our architecture and the other based on the paper. We perform a normalization before plugging in the U-Net architecture based on the range of RGB numbers, as shown in 8:

```
inputs = tf.keras.layers.Input((IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS))
s = tf.keras.layers.Lambda(lambda x: x / 255)(inputs)
```

Figure 8: Input Normalization

In the previous classification model, we have shown that the first convolutional block only needs 16 filters to perform well. We also add a random dropout in each convolution block to prevent the overfitting problem. Based on this experiment, we modified U-Net architecture as shown in Figure 9:



```python
# Downsampling
c1 = tf.keras.layers.Conv2D(16, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(s)
c1 = tf.keras.layers.Dropout(0.1)(c1)
c1 = tf.keras.layers.Conv2D(16, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c1)
p1 = tf.keras.layers.MaxPooling2D((2,2))(c1)

c2 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(p1)
c2 = tf.keras.layers.Dropout(0.1)(c2)
c2 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c2)
p2 = tf.keras.layers.MaxPooling2D((2,2))(c2)

c3 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(p2)
c3 = tf.keras.layers.Dropout(0.2)(c3)
c3 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c3)
p3 = tf.keras.layers.MaxPooling2D((2,2))(c3)

c4 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(p3)
c4 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c4)
c4 = tf.keras.layers.Dropout(0.2)(c4)
p4 = tf.keras.layers.MaxPooling2D((2,2))(c4)

c5 = tf.keras.layers.Conv2D(256, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(p4)
c5 = tf.keras.layers.Dropout(0.3)(c5)
c5 = tf.keras.layers.Conv2D(256, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c5)

# Upsampling
u6 = tf.keras.layers.Conv2DTranspose(128, (2,2), strides=(2,2), padding='same')(c5)
u6 = tf.keras.layers.concatenate([u6, c4])
c6 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(u6)
c6 = tf.keras.layers.Dropout(0.2)(c6)
c6 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c6)

u7 = tf.keras.layers.Conv2DTranspose(64, (2,2), strides=(2,2), padding='same')(c6)
u7 = tf.keras.layers.concatenate([u7, c3])
c7 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(u7)
c7 = tf.keras.layers.Dropout(0.2)(c7)
c7 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c7)

u8 = tf.keras.layers.Conv2DTranspose(32, (2,2), strides=(2,2), padding='same')(c7)
u8 = tf.keras.layers.concatenate([u8, c2])
c8 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(u8)
c8 = tf.keras.layers.Dropout(0.1)(c8)
c8 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c8)

u9 = tf.keras.layers.Conv2DTranspose(16, (2,2), strides=(2,2), padding='same')(c8)
u9 = tf.keras.layers.concatenate([u9, c1])
c9 = tf.keras.layers.Conv2D(16, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(u9)
c9 = tf.keras.layers.Dropout(0.1)(c9)
c9 = tf.keras.layers.Conv2D(16, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c9)

# Output
outputs = tf.keras.layers.Conv2D(1,(1,1), activation='sigmoid')(c9)
```

Figure 9: Modified U-Net Architecture

The original images have the size (448,448,3). We would keep this size for this network, but there is still a resizing procedure that allows this network to predict images with different sizes.

Hyperparameters of this network are listed in Table 3:

Table 3: U-Net Hyperparameters

Training	Validation frac in training	Testing	Optimizer	Loss Function	Metrics	Epochs

Training	**Validation frac in training**	**Testing**	**Optimizer**	**Loss Function**	**Metrics**	**Epochs**
107	10%	17	Adam	Binary Crossentropy	IoUScore	50

The training and testing losses are plotted in Figure 10:

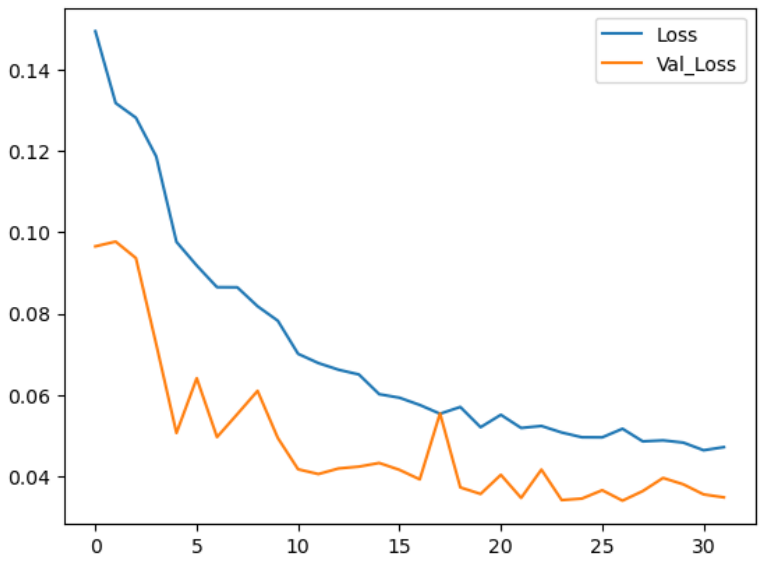


Figure 10: U-Net Loss Evaluation.

We can see that this trained model performs well on the testing data. It only takes 30 epochs because of the early-stop procedure.

Results and Analysis

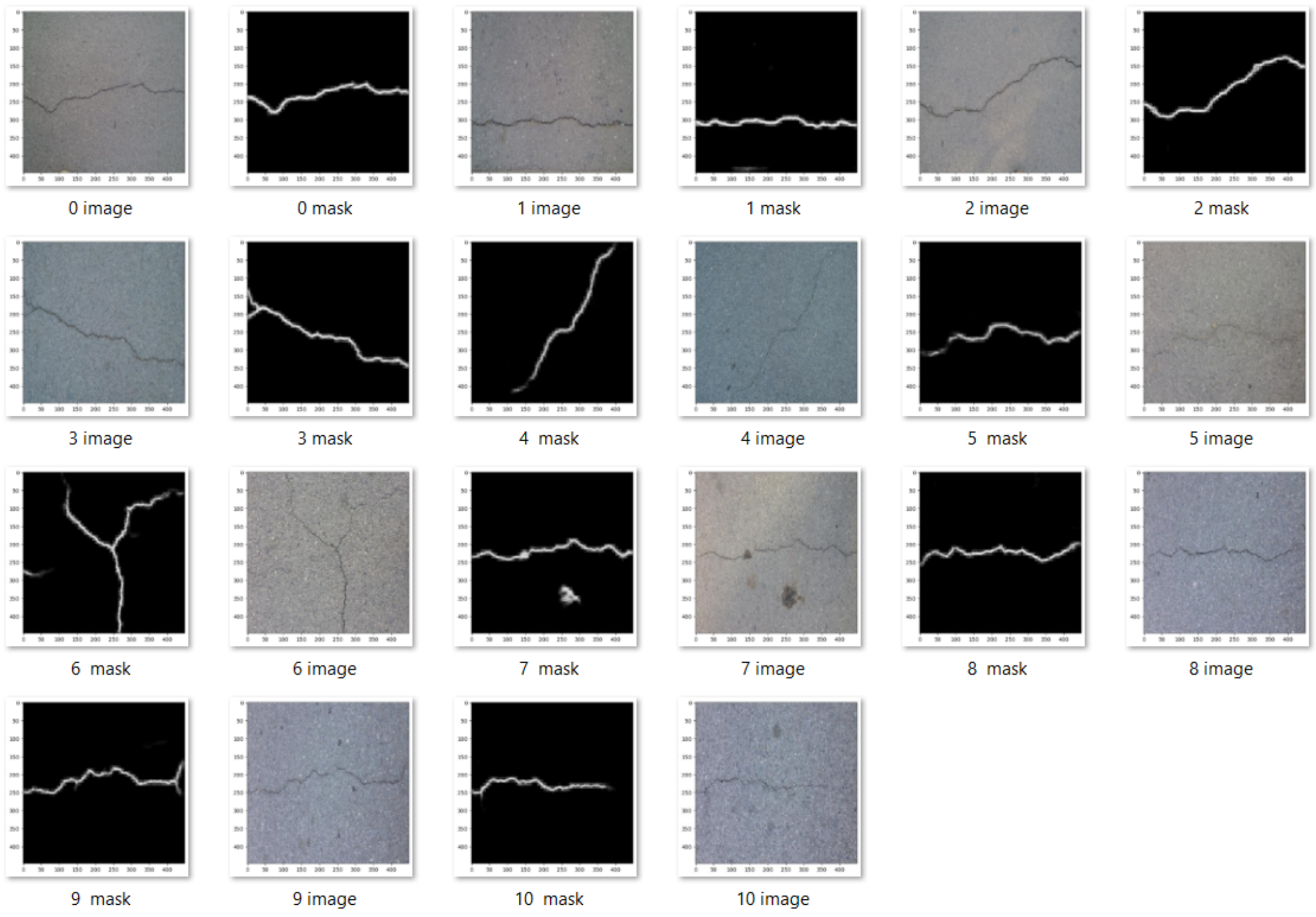


Figure 11: Prediction Result

The prediction results of our testing images are shown in Figure 11. It performs well. However, we can also notice that in testing image 7, the network recognizes the dark-colored zone with noise concentration (several disconnected noise regions close to each other) as a crack. The reason is that the network identifies cracks by detecting the edge of the color channels, which can be illustrated by predicting the crack from the RGB color wheel as shown in Figure 12:

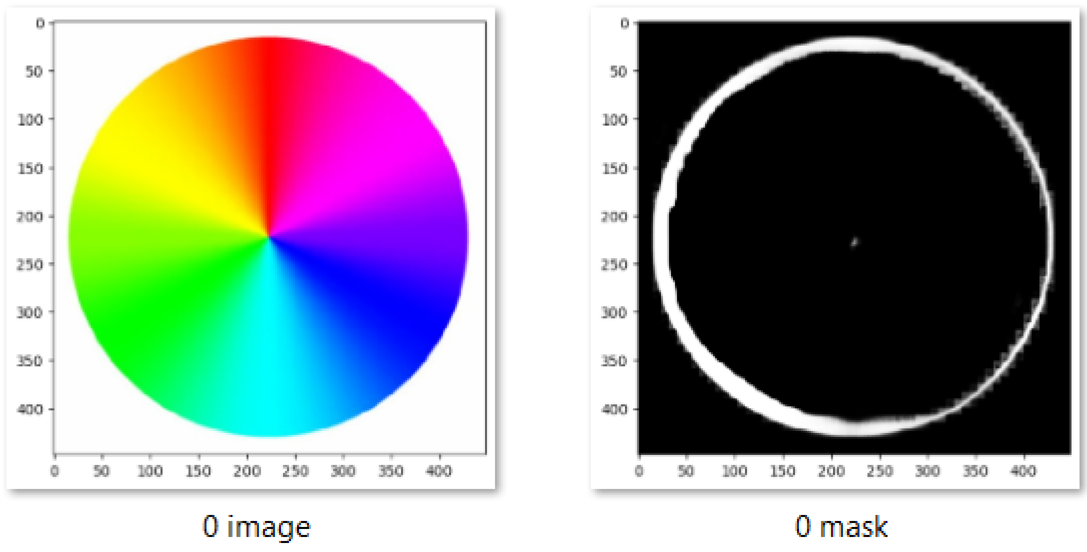


Figure 12: RGB Wheel Result

Several actions can be taken to avoid this noise concentration:

1. **Replace MaxPooling2D layers by AveragePooling2D.** AveragePooling2D could perform a flatten operation on its input. Rather than highlight the maximum point with MaxPooling2D, the network would generally benefit from this process.
2. **Build Up Preprocessor**
   a. *Normalize by Gray Scale.* There is a trade-off in the grayscale process: we will lose information during this procedure, but we can also take advantage as all the data is in the same stage with only one channel.
   b. *Multi-Kernel filters.* This process was inspired by the Inception network (the winner of the ImageNet Large Scale Visual Recognition Competition in 2014). The basic idea is to use multiple convolutional and pooling layers to extract more information from the input data.

The non-trainable Pooling layer is necessary to ensure the preprocessor works on our purpose. The origin image 7 and the trained image 7 obtained by the modified U-Net is shown in Figure 13:

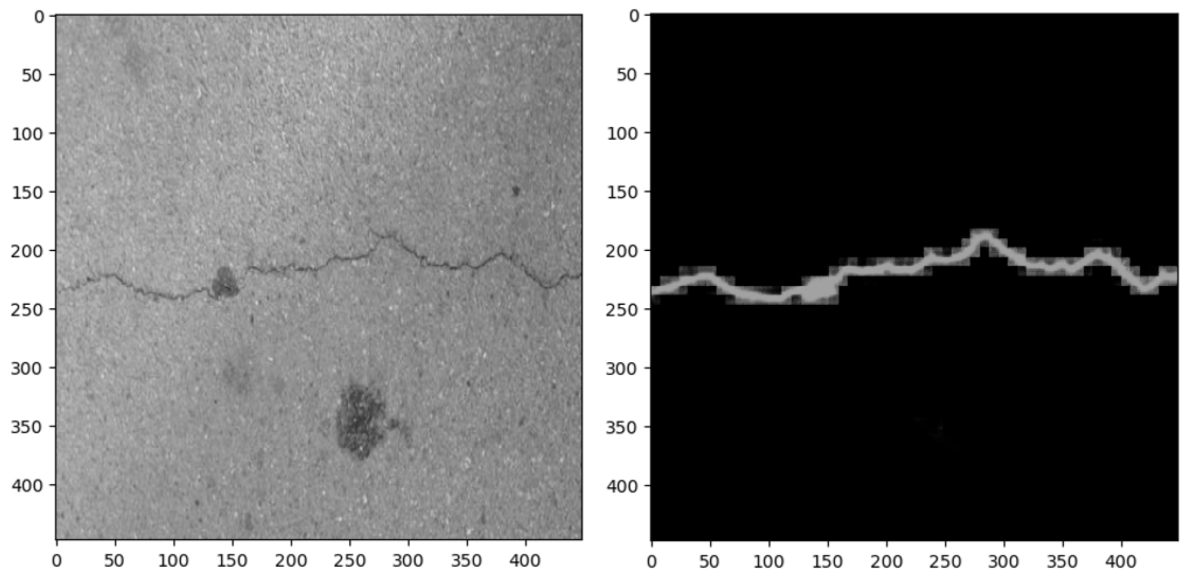


Figure 13: The Origin Image 7 and Image 7 Obtained by the Modified U-Net

Crack Segmentation on Rough Surfaces by Interception U-Net

Based on U-Net, we have trained the model that can label cracks. However, this model performs poorly in predicting images with bad-quality backgrounds (e.g., CRACK500, DeepCrack dataset, as shown in Figure 14). One reason is that the model was trained on the CFD dataset, which contains high-quality images with clear surfaces and obvious cracks, making the network over-trained. Network architecture is the other reason.

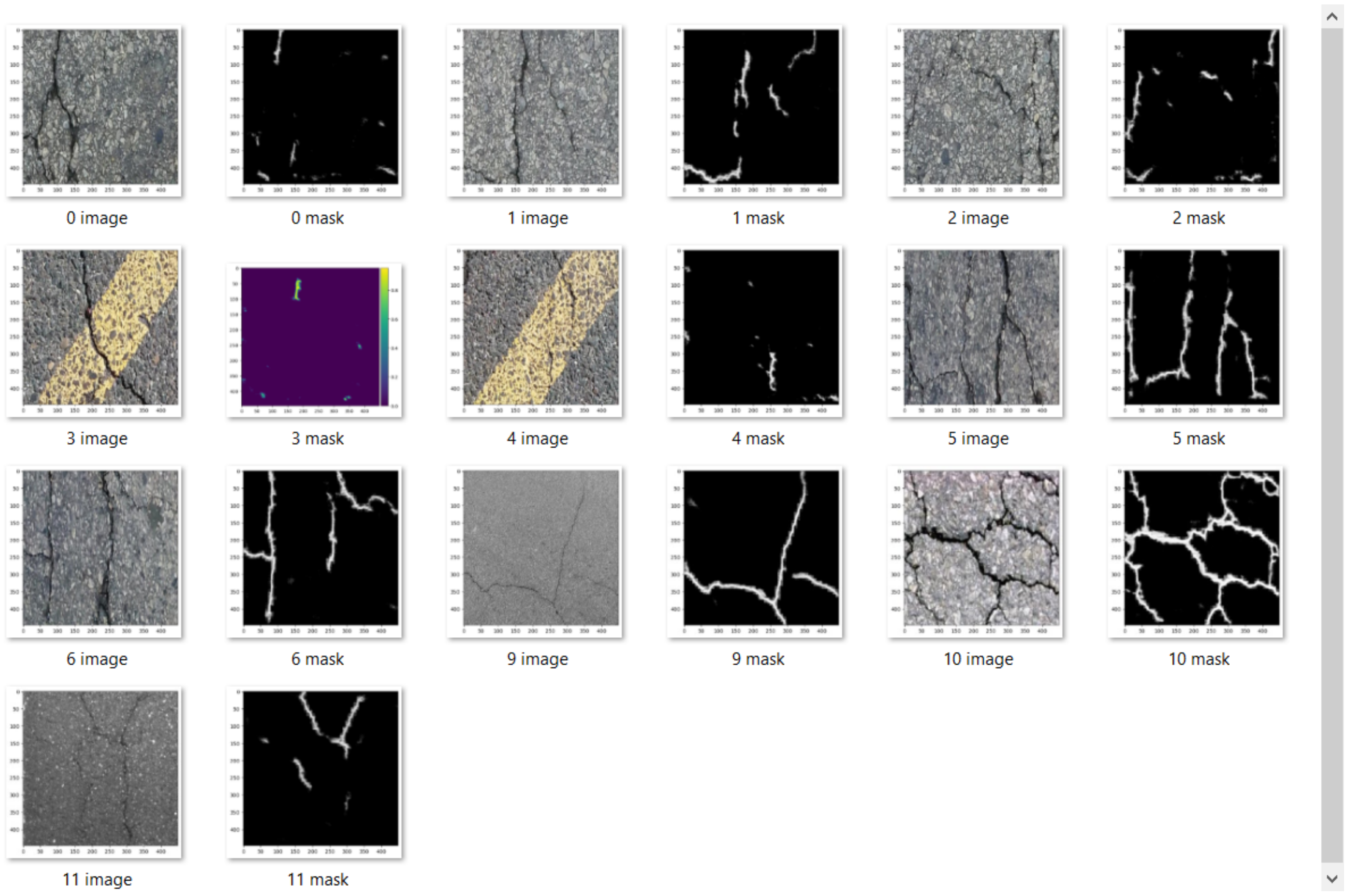


Figure 14: U-Net on Rough Surface Result

Instead of transfer the training to these datasets, we can design a network that is still trained on the CFD dataset and but also works well on CRACK500, DeepCrack, and other datasets with rough surfaces. Based on our previous work, we add an Inception network (as shown in 16) as our preprocessor and achieve good prediction on other test images. The new network is named "Inception U-Net", as the preprocessor is based on Inception and processed by U-Net. The segmentation results obtained by the new network is shown in Figure 15, and the hyperparameters of this network are listed in Table 4:

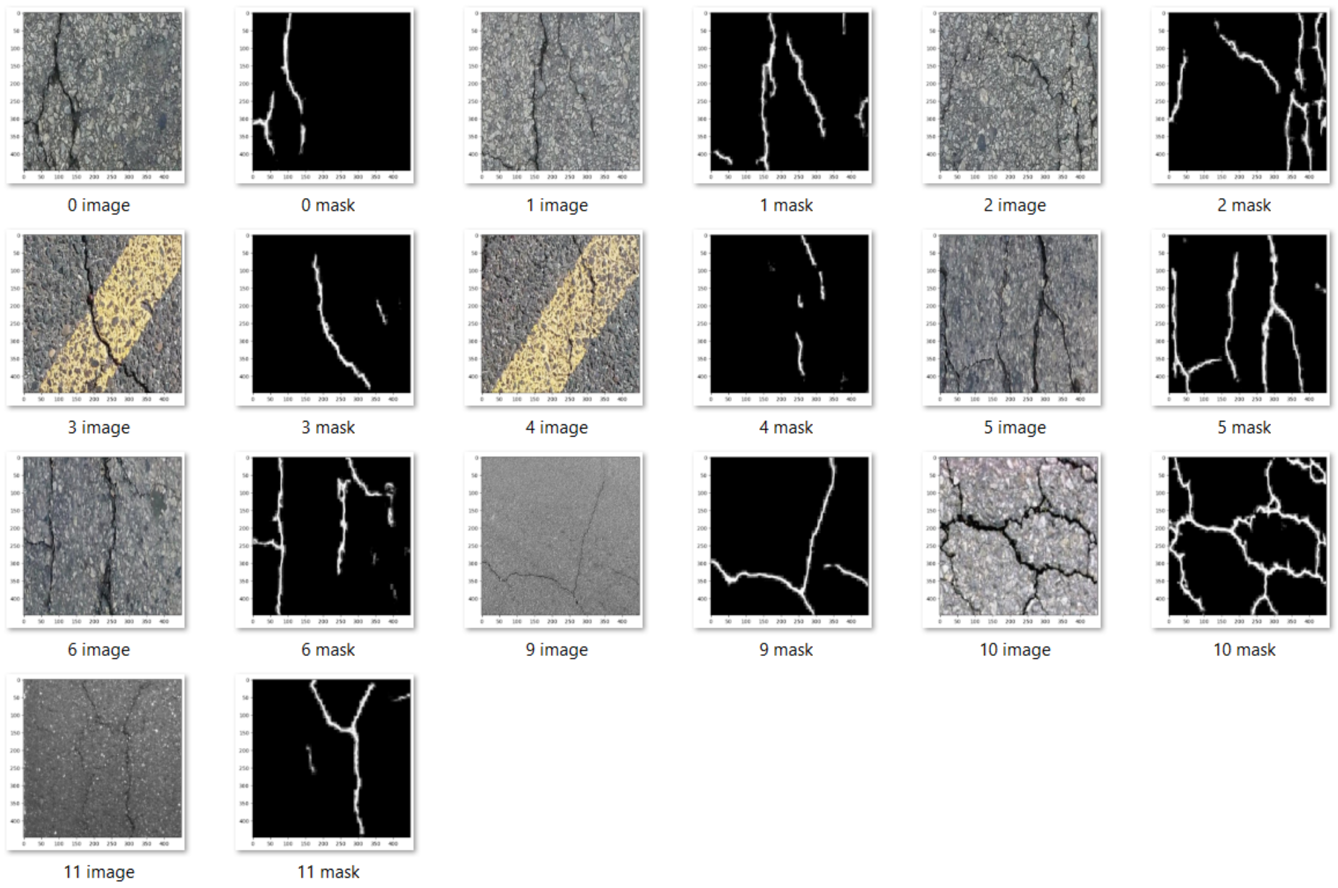
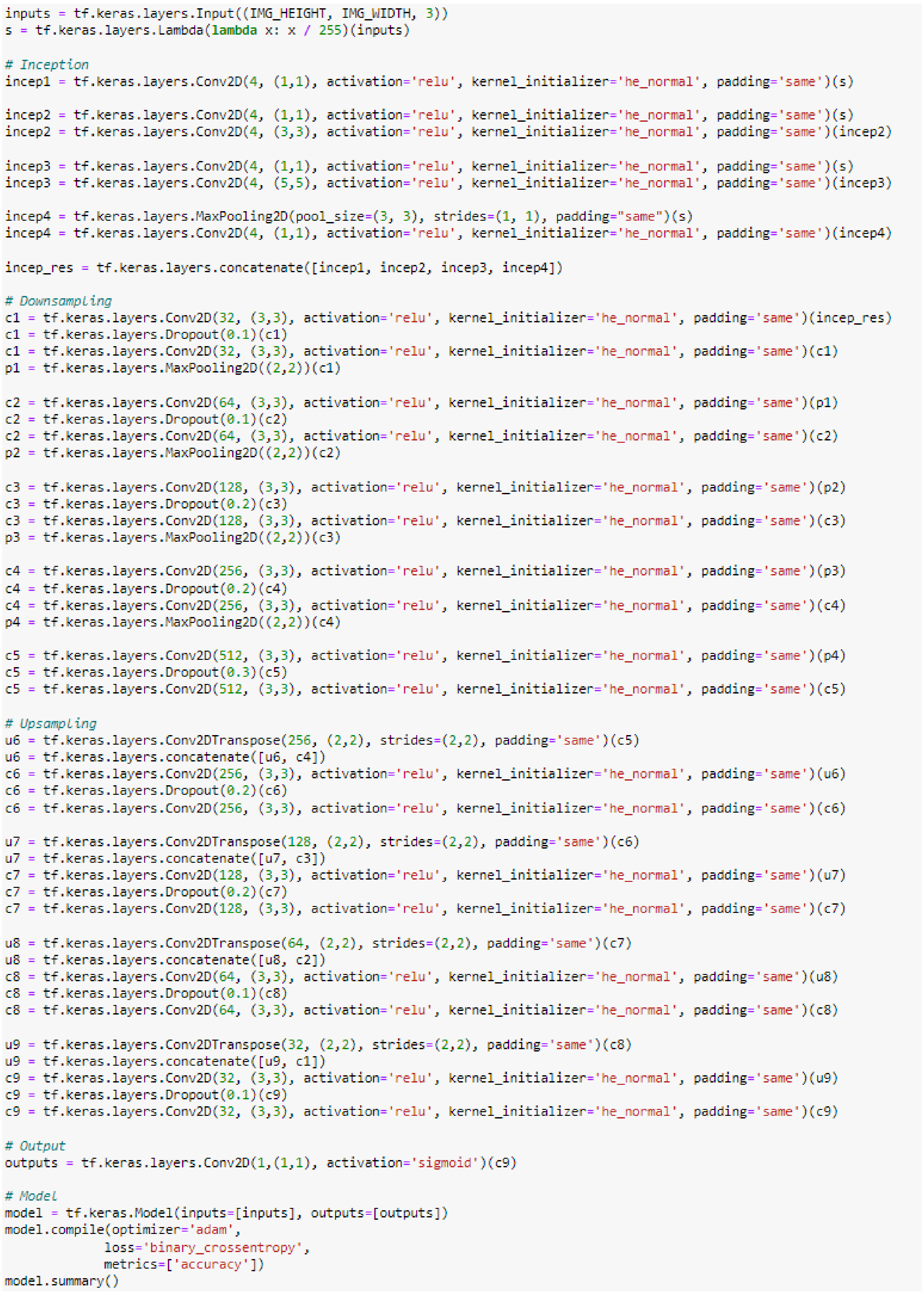


Figure 15: Inception U-Net on Rough Surface Result



```
inputs = tf.keras.layers.Input((IMG_HEIGHT, IMG_WIDTH, 3))
s = tf.keras.layers.Lambda(lambda x: x / 255)(inputs)

# Inception
incep1 = tf.keras.layers.Conv2D(4, (1,1), activation='relu', kernel_initializer='he_normal', padding='same')(s)

incep2 = tf.keras.layers.Conv2D(4, (1,1), activation='relu', kernel_initializer='he_normal', padding='same')(s)
incep2 = tf.keras.layers.Conv2D(4, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(incep2)

incep3 = tf.keras.layers.Conv2D(4, (1,1), activation='relu', kernel_initializer='he_normal', padding='same')(s)
incep3 = tf.keras.layers.Conv2D(4, (5,5), activation='relu', kernel_initializer='he_normal', padding='same')(incep3)

incep4 = tf.keras.layers.MaxPooling2D(pool_size=(3, 3), strides=(1, 1), padding="same")(s)
incep4 = tf.keras.layers.Conv2D(4, (1,1), activation='relu', kernel_initializer='he_normal', padding='same')(incep4)

incep_res = tf.keras.layers.concatenate([incep1, incep2, incep3, incep4])

# Downsampling
c1 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(incep_res)
c1 = tf.keras.layers.Dropout(0.1)(c1)
c1 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c1)
p1 = tf.keras.layers.MaxPooling2D((2,2))(c1)

c2 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(p1)
c2 = tf.keras.layers.Dropout(0.1)(c2)
c2 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c2)
p2 = tf.keras.layers.MaxPooling2D((2,2))(c2)

c3 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(p2)
c3 = tf.keras.layers.Dropout(0.2)(c3)
c3 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c3)
p3 = tf.keras.layers.MaxPooling2D((2,2))(c3)

c4 = tf.keras.layers.Conv2D(256, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(p3)
c4 = tf.keras.layers.Dropout(0.2)(c4)
c4 = tf.keras.layers.Conv2D(256, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c4)
p4 = tf.keras.layers.MaxPooling2D((2,2))(c4)

c5 = tf.keras.layers.Conv2D(512, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(p4)
c5 = tf.keras.layers.Dropout(0.3)(c5)
c5 = tf.keras.layers.Conv2D(512, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c5)

# Upsampling
u6 = tf.keras.layers.Conv2DTranspose(256, (2,2), strides=(2,2), padding='same')(c5)
u6 = tf.keras.layers.concatenate([u6, c4])
c6 = tf.keras.layers.Conv2D(256, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(u6)
c6 = tf.keras.layers.Dropout(0.2)(c6)
c6 = tf.keras.layers.Conv2D(256, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c6)

u7 = tf.keras.layers.Conv2DTranspose(128, (2,2), strides=(2,2), padding='same')(c6)
u7 = tf.keras.layers.concatenate([u7, c3])
c7 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(u7)
c7 = tf.keras.layers.Dropout(0.2)(c7)
c7 = tf.keras.layers.Conv2D(128, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c7)

u8 = tf.keras.layers.Conv2DTranspose(64, (2,2), strides=(2,2), padding='same')(c7)
u8 = tf.keras.layers.concatenate([u8, c2])
c8 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(u8)
c8 = tf.keras.layers.Dropout(0.1)(c8)
c8 = tf.keras.layers.Conv2D(64, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c8)

u9 = tf.keras.layers.Conv2DTranspose(32, (2,2), strides=(2,2), padding='same')(c8)
u9 = tf.keras.layers.concatenate([u9, c1])
c9 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(u9)
c9 = tf.keras.layers.Dropout(0.1)(c9)
c9 = tf.keras.layers.Conv2D(32, (3,3), activation='relu', kernel_initializer='he_normal', padding='same')(c9)

# Output
outputs = tf.keras.layers.Conv2D(1,(1,1), activation='sigmoid')(c9)

# Model
model = tf.keras.Model(inputs=[inputs], outputs=[outputs])
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()
```

Figure 16: Inception U-Net Structure

Table 4: U-Net Hyperparameters

Training	Validation frac in training	Testing	Optimizer	Loss Function	Metrics	Epochs
107	10%	17	Adam	Binary Crossentropy	IoUScore	100

We also put forward a hypothesis for crack segmentation: *MaxPooling2D works well for thin cracks but may fail for thick cracks, while AveragePooling plays a reverse role.*

Conclusion

In traditional CNN-based image crack-detection, people aim to train a network to tell a figure with or without cracks with simply two labels. However, how exactly such a network learns to recognize cracks is hard to measure. In this project, we reproduced CNN labeling images with or without cracks and modified and improved U-Net's ability to segment cracks. Results from U-Net tell much more about the location, shape, and length of cracks than mere labeling from CNN. We also tried to solve several potential issues in U-Net, including recognizing concentrated dark areas as cracks and performing badly on rough surface images. In practice, we also found that:

1. The model should not have extremely low loss to avoid overfitting.
2. MaxPooling2D works better for thin cracks, while AveragePooling works better for thick cracks.

Reproducible Work

The network codes, pre-trained models, and datasets for reproducible work are provided in the 'content.reproducible' folder.

Reference

1. **Surface Crack Detection** https://www.kaggle.com/datasets/arunrk7/surface-crack-detection

2. **U-Net: Convolutional Networks for Biomedical Image Segmentation**
Olaf Ronneberger, Philipp Fischer, Thomas Brox
Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015 (2015)
https://doi.org/gcgk7j
DOI: 10.1007/978-3-319-24574-4_28 · ISBN: 9783319245737

3. **Concrete Crack Conglomerate Dataset**
Eric Bianchi, Matthew Hebdon
University Libraries, Virginia Tech (2021) https://doi.org/gq9q35
DOI: 10.7294/16625056.v1