Concrete Crack Detection

Authors

- **Yu-Sian Lin**
  · ⬡ yslin0114
  Department of Civil and Environmental Engineering, University of Illinois

- **Yueh-Ti Lee**
  · ⬡ Yueh-Ti
  Department of Civil and Environmental Engineering, University of Illinois

- **Minjiang Zhu**
  · ⬡ Minjiang-Zhu
  Department of Civil and Environmental Engineering, University of Illinois

- **Chengyou Yue**
  · ⬡ TensorYue
  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 of the exact locations of cracks is smeared in the network. We aim to firstly rebuild a CNN-based model for crack detection, and then find and apply another model to locate the exact positions of cracks.

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 crack. Each class contains 20000 images. Examples are shown in Figure 1:

Negative   Positive

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

We then train a CNN-based network call U-Net [2] with the Concrete Crack Conglomerate Dataset [3] to segment the concrete from its cracks. The dataset contains over 10995 crack images (each has 448×448 pixels and RGB channels). Each image data has an origin image and a mask image. Examples are shown in Figure 2:
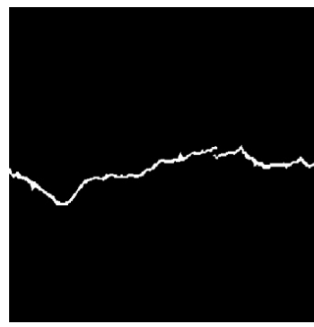
Figure 2: Origin and Mask Images from the Concrete Crack Conglomerate Dataset

Exploratory Data Analysis

In classification, the dataset consists of 20,000 images with cracks and 20,000 images without cracks. We are planning to use convolutional neural network to train a classification model. Figure 5

Figure 3. Number of Images with and without Cracks

In segmentation, we use the dataset consists of 21,996 images, splitting it into train data with 19,801 images and train data with 2,195 images. Here, we'll be applying U Net to train a model that is capable of pointing out the cracks in a concrete image. Figure 6

Figure 4. Number of Train Images and Test Images

Preliminary Predictive Modeling

Dataset:

- Concrete images with attached label 'Positive' or 'Negative'
- Concrete images and their masks

Goal:

- Concrete crack detection
- Concrete crack segmentation

Architecture:

- Concrete crack detection based on convolutional neural network (CNN)
- Concrete crack segmentation based on U-Net and improved U-Net using Inception as backbone
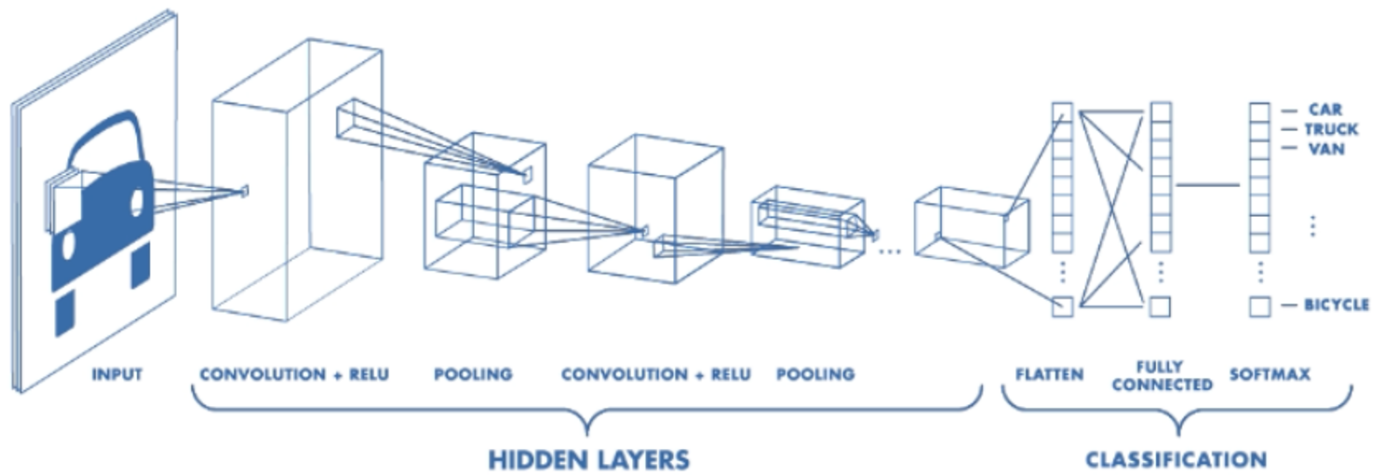
Concrete Crack Detection

Figure 3: Basic Convolutional Neural Network for Classification.

Modeling and training

Here we need to specify the hyperparameters of our layers:

1. Convolutional layer. Convolution is an operation that use filter to extract information that we want to detect.
   a. Filter size determines how many filters that would apply on the input tensor to produce the same number of channels, each filter could detect their specified geometries with colors in the given kernel size.
   b. Kernel size determines the size of the area the filters would apply.
   c. Activation function is added to achieve nonlinearity of the model, as the operation of filter is linear.
2. MaxPooling layer. MaxPooling is a down-sampling operation that allows our network to capture deeper information from original dimensions.
   a. Pool size and strides determine the dimensions of down-sample procedure.

In the first convolutional block, we would specify 16 filters (consider 8 straight lines and 8 curves) with 3 by 3 kernel (consider the size of crack is relatively small) and ReLu as activation function in the Conv2D layer and go deeper with pool size of 2 by 2 in the MaxPool2D layer.

And in the second convolutional block, we double the channels number to 32 with Conv2D, apply the same MaxPool2D.

Finally, we process our data input an GlobalAveragePooling2D layer (because we are focusing on the whole image, not a part of it in segmentation) and dense the tensor to 1 with Sigmoid function (As we only have 1 class of result, for multiclass classification, use SoftMax).

The result of our network would be a number in (0,1), we could treat this as a probability of crack in the image.

```
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.

Notice that the original images have size of (227,227,3), we would resize it into (120,120,3) in preprocess procedure.

```
_____
Layer (type)                Output Shape              Param #
=================================================================
input_1 (InputLayer)        [(None, 120, 120, 3)]     0

conv2d (Conv2D)             (None, 118, 118, 16)      448

max_pooling2d (MaxPooling2D  (None, 59, 59, 16)       0
)

conv2d_1 (Conv2D)           (None, 57, 57, 32)        4640

max_pooling2d_1 (MaxPooling  (None, 28, 28, 32)       0
2D)

global_average_pooling2d (G  (None, 32)               0
lobalAveragePooling2D)

dense (Dense)               (None, 1)                 33

=================================================================
Total params: 5,121
Trainable params: 5,121
Non-trainable params: 0
_____
```

Figure 5: Model Summary and_Hyperparameters.

Table 1: Hyperparameters.

training	validation	testing	Optimizer	Loss Function	Metrics	Epochs
3360	840	1800	Adam	Binary Crossentropy	Accuracy	100

Prediction and Analysis

We can check our trained model by inputting the testing data, below is the confusion matrix our model generated:
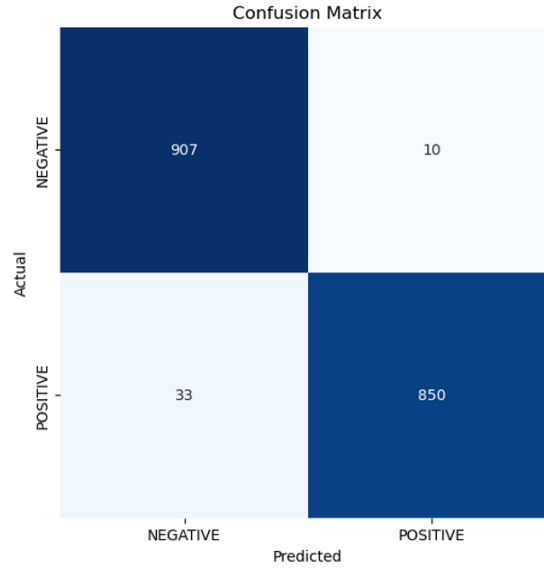
Figure 6: **Confusion Matrix.**
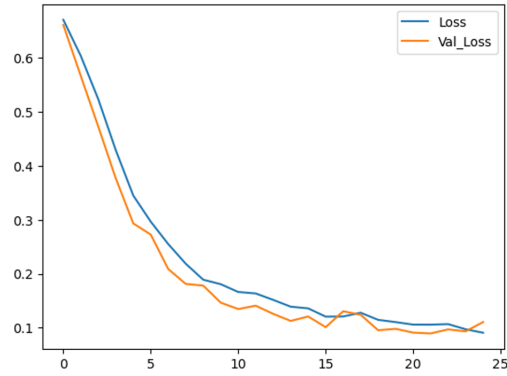
Figure 7: **Loss Evolution.**

We can see that this trained model has good performance on the testing data, it only takes 25 epochs because of our early stop condition. We can further calculate the following parameters:

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

Table 2: Result Analysis

	Precision	Recall	Score	Support
Negative	0.96	0.99	0.98	917
Positive	0.99	0.96	0.97	883

	Precision	Recall	Score	Support
Macro Avg	0.98	0.97	0.97	1800
Weighted Avg	0.97	0.97	0.97	1000

It has overall good performance.

Concrete Segmentation

The original U-Net architecture is shown below:
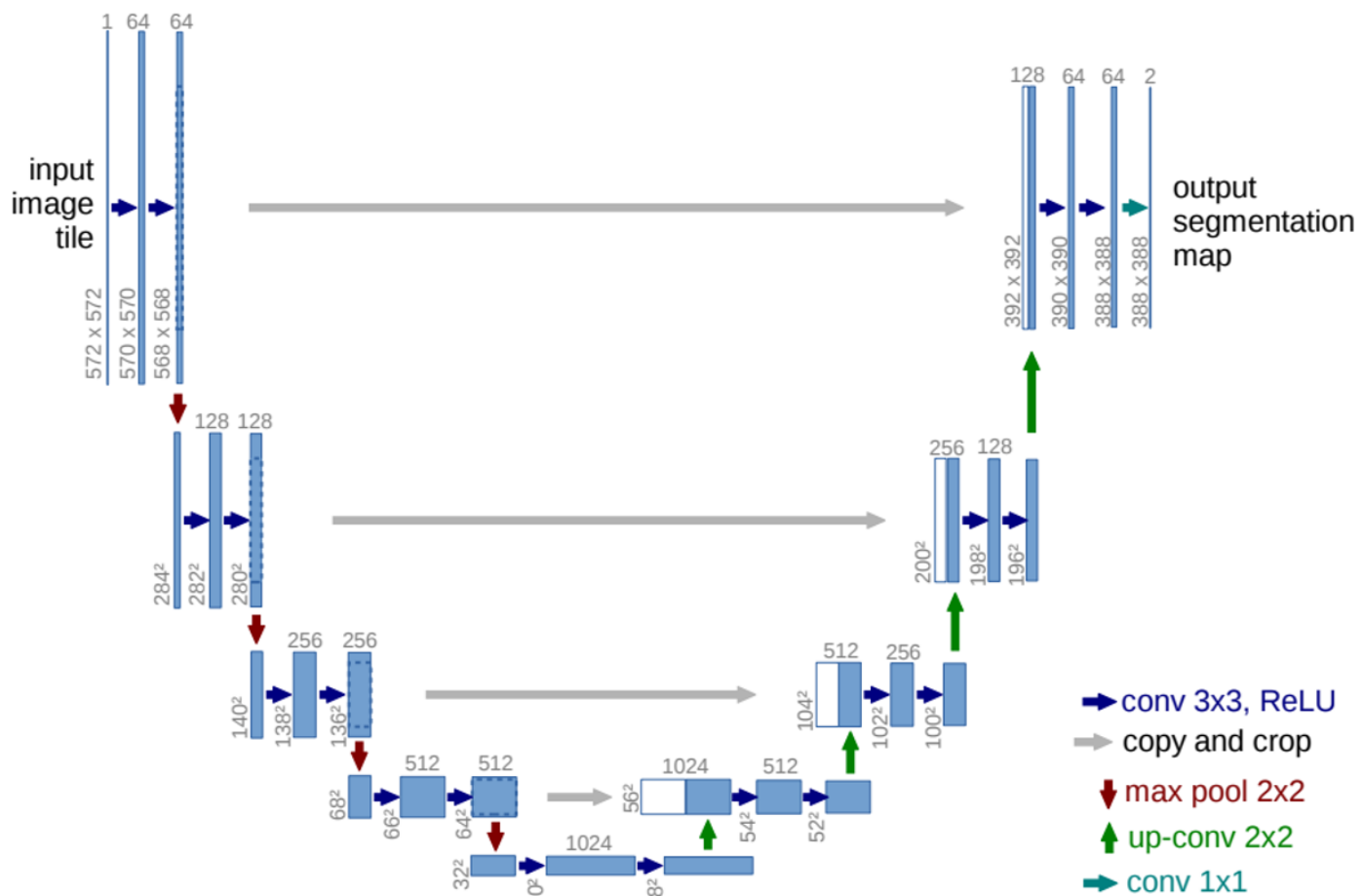
Figure 8: U-net Architecture.

Modeling and Training

For achieving segmentation, we have built and trained two models, one based on our own architecture, and the other based on the paper. We perform a normalization before plug in the U-Net architecture based on the range of RGB number:

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

Figure 9: Input Normalization.

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

```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 10: Modified U-Net Architecture.

Notice that the original images have size of (448,448,3), we would keep this size for this network, but there is still a resize procedure that allow this network to predict images with different size.

```
Layer (type)                  Output Shape          Param #    Connected to              concatenate_4 (Concatenate)   (None, 56, 56, 256)   0      ['conv2d_transpose_4[0][0]',
===================================================================================                                                            'dropout_12[0][0]']
input_2 (InputLayer)          [(None, 448, 448, 3   0          []
                               )]                                                          conv2d_29 (Conv2D)            (None, 56, 56, 128)    295040 ['concatenate_4[0][0]']
lambda_1 (Lambda)             (None, 448, 448, 3)   0          ['input_2[0][0]']           dropout_14 (Dropout)          (None, 56, 56, 128)    0      ['conv2d_29[0][0]']

conv2d_19 (Conv2D)            (None, 448, 448, 16   448        ['lambda_1[0][0]']          conv2d_30 (Conv2D)            (None, 56, 56, 128)    147584 ['dropout_14[0][0]']
                               )
                                                                                           conv2d_transpose_5 (Conv2DTran (None, 112, 112, 64  32832  ['conv2d_30[0][0]']
dropout_9 (Dropout)           (None, 448, 448, 16   0          ['conv2d_19[0][0]']         spose)                         )
                               )
                                                                                           concatenate_5 (Concatenate)   (None, 112, 112, 12   0      ['conv2d_transpose_5[0][0]',
conv2d_20 (Conv2D)            (None, 448, 448, 16   2320       ['dropout_9[0][0]']                                        8)                            'conv2d_24[0][0]']
                               )
                                                                                           conv2d_31 (Conv2D)            (None, 112, 112, 64    73792  ['concatenate_5[0][0]']
max_pooling2d_4 (MaxPooling2D) (None, 224, 224, 16   0          ['conv2d_20[0][0]']                                       )
                               )
                                                                                           dropout_15 (Dropout)          (None, 112, 112, 64    0      ['conv2d_31[0][0]']
conv2d_21 (Conv2D)            (None, 224, 224, 32   4640       ['max_pooling2d_4[0][0]']                                  )
                               )
                                                                                           conv2d_32 (Conv2D)            (None, 112, 112, 64    36928  ['dropout_15[0][0]']
dropout_10 (Dropout)          (None, 224, 224, 32   0          ['conv2d_21[0][0]']                                       )
                               )
                                                                                           conv2d_transpose_6 (Conv2DTran (None, 224, 224, 32  8224   ['conv2d_32[0][0]']
conv2d_22 (Conv2D)            (None, 224, 224, 32   9248       ['dropout_10[0][0]']        spose)                         )
                               )
                                                                                           concatenate_6 (Concatenate)   (None, 224, 224, 64   0      ['conv2d_transpose_6[0][0]',
max_pooling2d_5 (MaxPooling2D) (None, 112, 112, 32   0          ['conv2d_22[0][0]']                                       )                             'conv2d_22[0][0]']
                               )
                                                                                           conv2d_33 (Conv2D)            (None, 224, 224, 32    18464  ['concatenate_6[0][0]']
conv2d_23 (Conv2D)            (None, 112, 112, 64   18496      ['max_pooling2d_5[0][0]']                                  )
                               )
                                                                                           dropout_16 (Dropout)          (None, 224, 224, 32    0      ['conv2d_33[0][0]']
dropout_11 (Dropout)          (None, 112, 112, 64   0          ['conv2d_23[0][0]']                                       )
                               )
                                                                                           conv2d_34 (Conv2D)            (None, 224, 224, 32    9248   ['dropout_16[0][0]']
conv2d_24 (Conv2D)            (None, 112, 112, 64   36928      ['dropout_11[0][0]']                                       )
                               )
                                                                                           conv2d_transpose_7 (Conv2DTran (None, 448, 448, 16  2064   ['conv2d_34[0][0]']
max_pooling2d_6 (MaxPooling2D) (None, 56, 56, 64)    0          ['conv2d_24[0][0]']         spose)                         )

conv2d_25 (Conv2D)            (None, 56, 56, 128)   73856      ['max_pooling2d_6[0][0]']   concatenate_7 (Concatenate)   (None, 448, 448, 32   0      ['conv2d_transpose_7[0][0]',
                                                                                                                         )                             'conv2d_20[0][0]']
conv2d_26 (Conv2D)            (None, 56, 56, 128)   147584     ['conv2d_25[0][0]']
                                                                                           conv2d_35 (Conv2D)            (None, 448, 448, 16    4624   ['concatenate_7[0][0]']
dropout_12 (Dropout)          (None, 56, 56, 128)   0          ['conv2d_26[0][0]']                                       )

max_pooling2d_7 (MaxPooling2D) (None, 28, 28, 128)   0          ['dropout_12[0][0]']        dropout_17 (Dropout)          (None, 448, 448, 16    0      ['conv2d_35[0][0]']
                                                                                                                         )
conv2d_27 (Conv2D)            (None, 28, 28, 256)   295168     ['max_pooling2d_7[0][0]']
                                                                                           conv2d_36 (Conv2D)            (None, 448, 448, 16    2320   ['dropout_17[0][0]']
dropout_13 (Dropout)          (None, 28, 28, 256)   0          ['conv2d_27[0][0]']                                       )

conv2d_28 (Conv2D)            (None, 28, 28, 256)   590080     ['dropout_13[0][0]']        conv2d_37 (Conv2D)            (None, 448, 448, 1)    17     ['conv2d_36[0][0]']

conv2d_transpose_4 (Conv2DTran (None, 56, 56, 128)  131200     ['conv2d_28[0][0]']        ===================================================================================
spose)                                                                                    Total params: 1,941,105
                                                                                          Trainable params: 1,941,105
                                                                                          Non-trainable params: 0
```

Figure 11: Model Summary

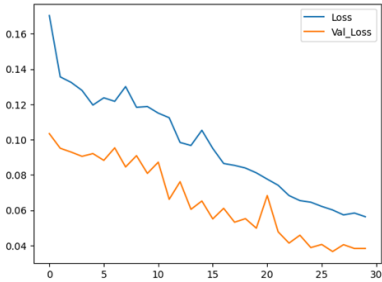U-Net could learn from very few numbers of dataset, so we only need to have a small dataset.

Figure 12: U-Net Loss Evaluation.

Table 3: U-Net Hyperparameters.

training	validation	testing	Optimizer	Loss Function	Metrics	Epochs
105	2	17	Adam	Binary Crossentropy	Accuracy	50

We can see that this trained model has good performance on the testing data, it only takes 30 epochs because of our early stop condition.
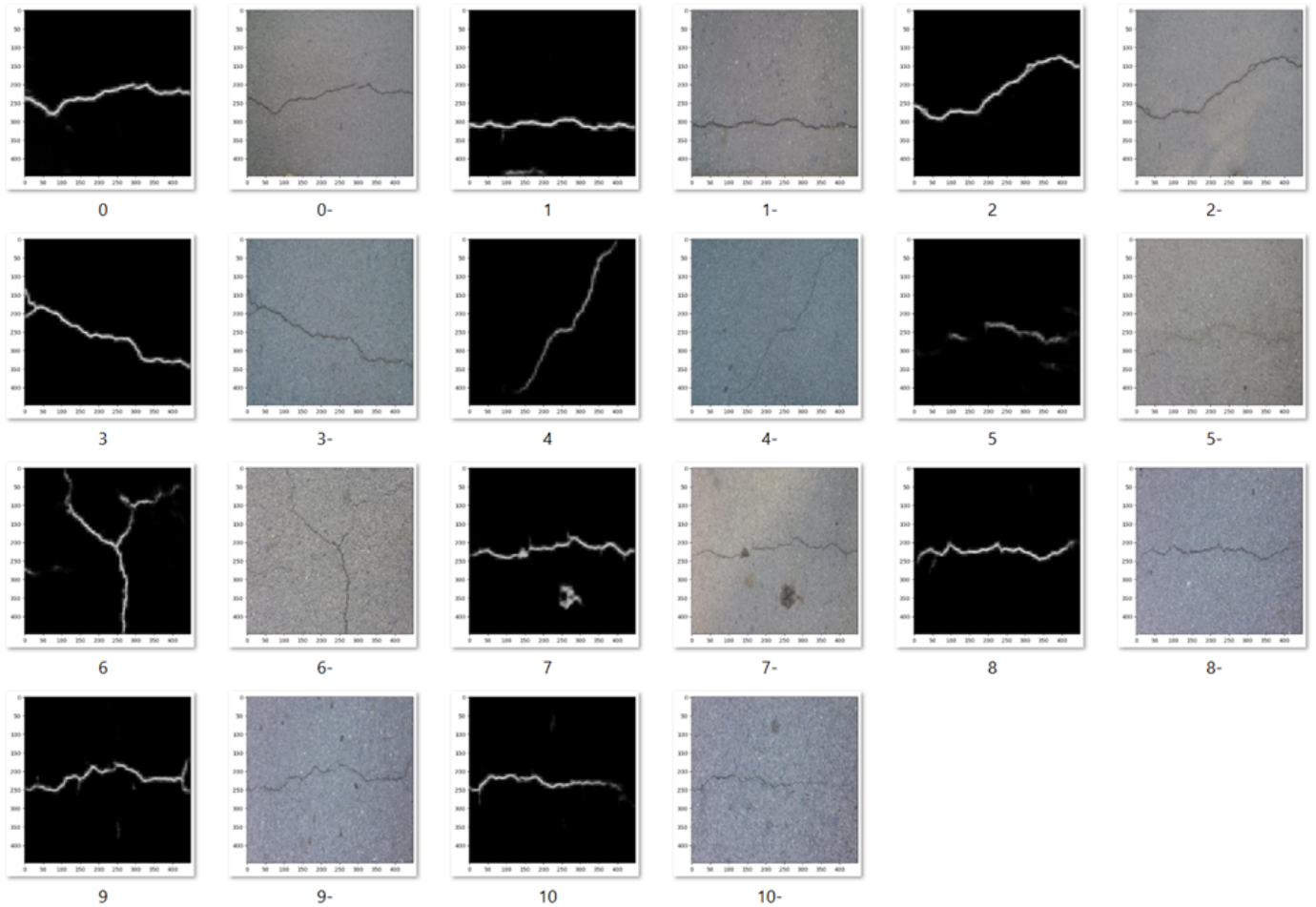
Prediction and Analysis

Figure 13: Prediction Result

Here are the prediction results of our testing images, it performs extremely well.

We could also notice that in the testing image 7, the network predict zone with dark color as crack. It is because the filter would not only capture geometries of the image in the kernel, but also the color of it (recall we have RGB channels as the input).

We are still working on the next U-Net model with Conv2D=>BatchNormalization=>ReLU as conv_block! We also trained a U-Net model with pretrained Inception as backbone, but we haven't upload it yet!

Computer vision-based concrete crack detection using U-net fully convolutional networks [4]

U-Net: Convolutional Networks for Biomedical Image Segmentation [2]

Surface Crack Detection | Kaggle [1].

Concrete Crack Conglomerate Dataset | Virginia Tech [3]

Performance Evaluation of Deep CNN-Based Crack Detection and Localization Techniques for Concrete Structures [5]

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

4. **Computer vision-based concrete crack detection using U-net fully convolutional networks**
   Zhenqing Liu, Yiwen Cao, Yize Wang, Wei Wang
   Automation in Construction (2019-08) https://doi.org/gghznx
   DOI: 10.1016/j.autcon.2019.04.005

5. **Performance Evaluation of Deep CNN-Based Crack Detection and Localization Techniques for Concrete Structures**
   Luqman Ali, Fady Alnajjar, Hamad Al Jassmi, Munkhjargal Gocho, Wasif Khan, MAdel Serhani
   Sensors (2021-03-01) https://doi.org/gm8k7f
   DOI: 10.3390/s21051688 · PMID: 33804490 · PMCID: PMC7957757