DEVELOPMENT OF A METHOD FOR AUTOMATED CATEGORIZATION OF DEFECTS ON NATURAL STONE FACADES OF BUILDINGS ON THE BASIS OF AN APPROPRIATE DEEP LEARNING APPROACH

ERARBEITUNG EINES VERFAHRENS ZUR AUTOMATISIERTEN KATEGORISIERUNG VON SCHÄDEN AN NATURSTEINFASSADEN VON BAUWERKEN UNTER EINSATZ EINES GEEIGNETEN DEEP LEARNING VERFAHRENS

Diploma Thesis

Approved by the Faculty of Civil Engineering
Institut of construction informatics
Dresden University of Technology
Dresden, Germany

Written by
Jiesheng Yang

Supervisors:
Dr. -Ing. Peter Katranuschkov
M. Sc. Fangzheng Lin
Dr. -Ing. Sebastian Fuchs
Dipl. -Math. Robert Schülbe

Date of submission: 15. May 2019

Declaration

Hereby, I declare that the diploma thesis report entitled " DEVELOPMENT OF A A METHOD FOR AUTOMATED CATEGORIZATION OF DEFECT ON NATURAL STONE FACADES OF BUILDINGS ON THE BASIS OF AN APPROPRIATE DEEP LEARNING APPROACH" is carried out independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted.

Dresden, 13.05.2019

Jiesheng Yang

Abstract

To improve the efficiency of the renovation process, this thesis proposed a convolutional neural network based method to detect cracks of nature stones. At first, a dataset of concrete crack with 40,000 images in two classes was built. After analyzing of classic CNN structures, we built a network structure and trained it to detect concrete cracks. With the method of transfer learning, the model that is able to detect cracks of nature stones is also trained. Besides, we trained a model that can distinguish cracks from joints of nature stones.

Contents

List of Tables

List of Figures

Chapter 1

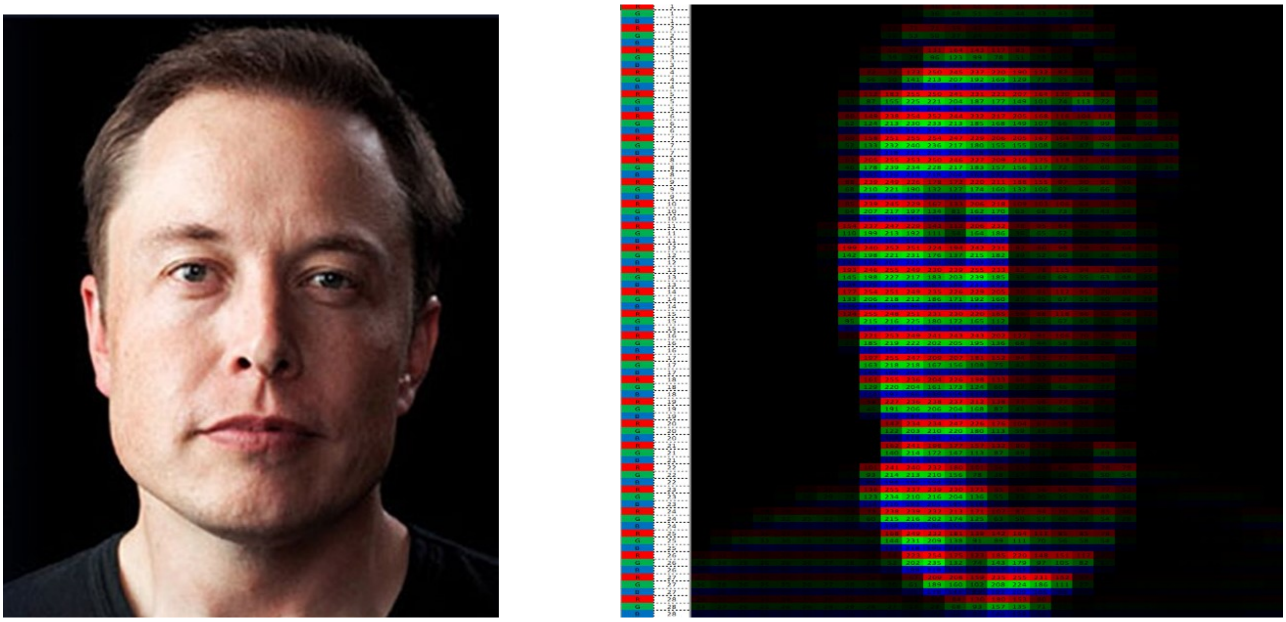
Introduction

1.1 Background

In the past decades, there is a growing demand for maintenance and renovation works for buildings made of natural stone facades. Therefore, some emphasis is placed on the need to reduce the construction costs as well as on improving the efficiency of the renovation process.[31] For this purpose, an identification and classification method for the defects occurring in natural stone using image recognition is very promising.

Presently, the inspection and diagnoses of the defects in stonework are mostly done by special equipment and human work by means of ultrasonic techniques, thermal techniques and assisted visual analysis.[27] However, issues from the high expense and the intensive human labor cost have not been adequately addressed.

Image recognition is one of the skills that humans get from the very first moment we are born and it is gained naturally and easily for adults. Generalized from our prior knowledge, human beings are able to recognize patterns and objects quickly even in different image environments. However, we didn't share this skill with machines. While human and animal brains recognize objects with ease, computers have difficulty with the task. Image recognition, in the context of machine vision, is a process of extracting meaningful information from given images, such as the feature of objects. Computers can use the Deep Learning (DL) to achieve image recognition.



(a) Human vision (b) Computer vision

Figure 1.1: Human vision and computer vision [33]

Instead of a human face with resolution of 28×28 pixels, computers see only an array of pixel values (see Figure 1.1). Depending on the resolution and the size of images, it breaks the image into a $28 \times 28 \times 3$ matrix of pixels[1] and stores the value of color for each pixel at the representative points. These values, from 0 to 255 at each position describing the pixel intensity, are the only inputs available to the computer.

The idea of image recognition is that with the matrix of numbers, a computer outputs values which describe the probability of Figure 1.1 (a) being which class (e.g. 0.85 for Elon Musk , 0.15 for Robert Downey Jr, 0.05 for Steve Jobs).

Image recognition is used to perform a large number of machine-based visual tasks, such as labeling the content of images with meta-tags, performing image content search and guiding autonomous robots, self-driving cars and accident avoidance systems.[14]

Until now, considerable researches have been conducted detecting damages by vision-based methods, primarily using Image Processing Techniques (IPTs) which has been proposed to

[1] The 3 refers to three color channels R, G, B.

redeem the situation. One significant advantage of IPTs is that almost all superficial defects (e.g. cracks and corrosion) are likely identifiable.[15]

An early comparative study using four edge detection methods (the fast Haar transform (FHT), fast Fourier transform, Sobel edge detector, and Canny edge detector) was conducted by Abdel-Qader to find concrete cracks. He proved that FHT is a better approach to solve the problem. After this study Yeum and Dyke (2015) used IPTs combined with a sliding window technique to detect steel cracks.

However, the results of edge detection are mainly affected by the noises of the image. Several factors in the real-world, including low light conditions, camera sensor size, higher ISO settings, and long exposures, can affect the level of noise.[5] To deal with the challenge of noises from the real-world, an approach using IPT-based image feature extractions and DL algorithms based classifications is implemented.

For different research purposes and verity of industrial fields applications, many types of artificial neural networks (ANNs) have been developed. For example, convolutional neural networks (CNNs) show great promise at image recognition. 2012 is the first year as Alex Krizhevsky used CNN to win that year's ImageNet competition. This competition is equivalent to the Olympics game in computer vision world. His AlexNet achieves an astounding improvement by dropping a top 5 error[2] from 26% to 15.4%. Inspired by the visual cortex of animals, CNN is now a state-of-the-art technique in image recognition challenges. In contrast to the standard NN, CNN can reduce much more computational effort, since it has the sparsely connected neurons and the pooling process. Furthermore, it works as an automatic feature extractor without bother users with feature selection or confused users whether the extracted feature would work with the model or not. CNN extracts all possible features, from the low-level ones like edges, to the higher-level features like faces and objects.

As a specialized sub-field of machine learning, DL is large neural networks that keeps getting better as you feed them more data. *When you hear the term DL, just think of a large deep*

[2] Top 5 error is the rate at which, given an image, the model does not output the correct label with its top 5 predictions.

neural net. Deep refers to the number of layers typically and so this kind of the popular term that's been adopted in the press. I think of them as deep neural networks generally.[19]

In DL, a computer model can learn to perform classification tasks directly from videos, images, texts, or sound. DL models have proven promising and very useful. It can achieve a very high accuracy, sometimes even exceed human-level performance. The models are usually trained by using a large set of labeled data and neural network architectures that contain many layers.[13]

Therefore, a vision-based method using a deep architecture of CNN for detecting cracks and other natural stone defects is proposed.

It should be pointed out that the train speed accelerates with the increasing computational power of GPU. Therefore, DL has great developmental potential and wide industrial application prospect.

1.2 Research Goals

The research goals of this thesis is to use CNN to build a nature stone cracks detector from image inputs, and to prove the applicability of CNN for more scenarios in building renovation.The main contributions in this work are as follows:

- The concrete dataset which contains 40,000 images in two classes is made to train the network.
- The CNN structure for crack detection is built after analysis of classic CNN structures.
- A model is trained to be able to detect concrete cracks.
- After building a small dataset of nature stone crack with 150 images, a model is trained with the method of transfer learning and is able to detect cracks of nature stones.
- A dataset including 150 images classed with joints and cracks of nature stone is made. With transfer learning, a model is trained and is able to distinguish cracks from joints.

1.3 Outline

The structure of this thesis is described as follows:

Chapter 2 presents the state of the art of present studies on CNN. To build our own network structure, roles of different layers and the theory behind them are explained.

Chapter 3 gives a brief introduction of nature stone defects.

Chapter 4 describes in detail how we build databanks and our network structures, and how to tune the hyperparameters and train the models. After training, two models that detect concrete cracks and nature stone cracks are built.

Chapter 5 goes a further step to apply CNN on nature stone problems. A model that distinguishes joints and cracks of nature stones is built.

Chapter 6 presents the training results and evaluates the trained models.

Chapter 7 deals with discussions of our findings, where some limitations of proposed method are also addressed.

Chapter 8 summarize this work and gives suggestions for future work.

Chapter 2

Methology

For the purpose of building our own CNN architecture, we will at first review classic CNN architectures and then elaborate the role of each layer. After having a basic understanding, the CNN architecture for crack detection will be build on the end of this chapter.

2.1 Classic Architecture

A Convolutinonal Neural Network (CNN) is very similar to ordinary ANNs, they both use neurons with learnable weights and biases. The whole network takes images as inputs and then outputs the class scores on the end. Unlike primitive hand-engineered filters, CNNs can be trained and are capable of learning these filters.[1][29]

The ImageNet project is an image database which can be a useful resource for researchers, educators, students and everyone that want to commit themselves into image studies.[7] To make researchers across the world have a chance to present and compare their newest efforts, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is held annually since 2010. This challenge evaluates the algorithms for object detection and image classification at large scale. In the following text, we will see the top competitors of the CNN architectures.

LeNet-5 is a classic CNN architecture proposed by Yann LeCun, Leon Bottou, Yosuha Bengio

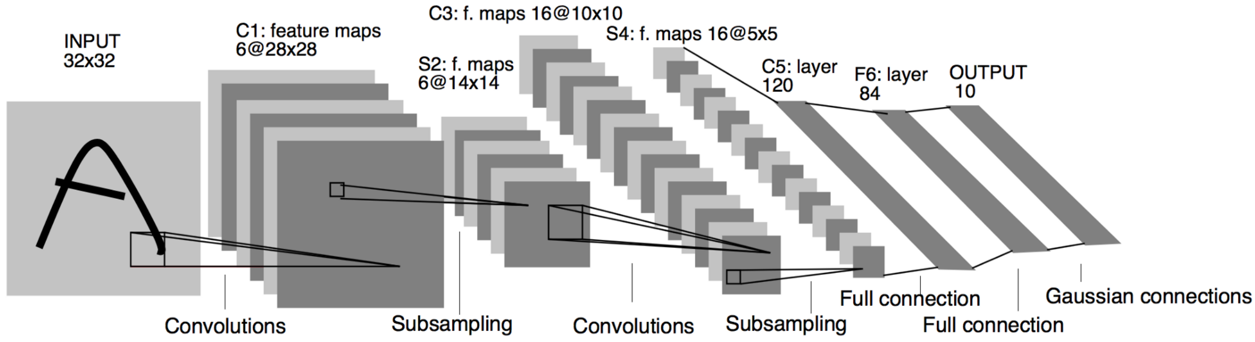


Figure 2.1: LeNet-5 architecture [23]

and Patrick Haffner in 1998.[23] It is applied in banking to recognize handwritten numbers on checks. Because of the limited computing power at that time, grayscale images in 32×32 pixel is considered as inputs. LeNet-5 has 7 layers, 3 of them are convolutional layers. This structure has around 60 thousand parameters.

In 2012 Alex Krizhevsky beats out all the prior competitors and won the LSVRC with **AlexNet**.[22] This CNN architecture reduces the top-5 error from 26% (XRCE) to 15.3%. It uses ReLU activation function instead of Sigmoid or Tanh functions for the same accuracy but with a five fold speed. AlexNet is much deeper compared with LeNet-5 and has complexer structure and stacked convolutional layers. Because of that, this structure has around 60 million parameters.

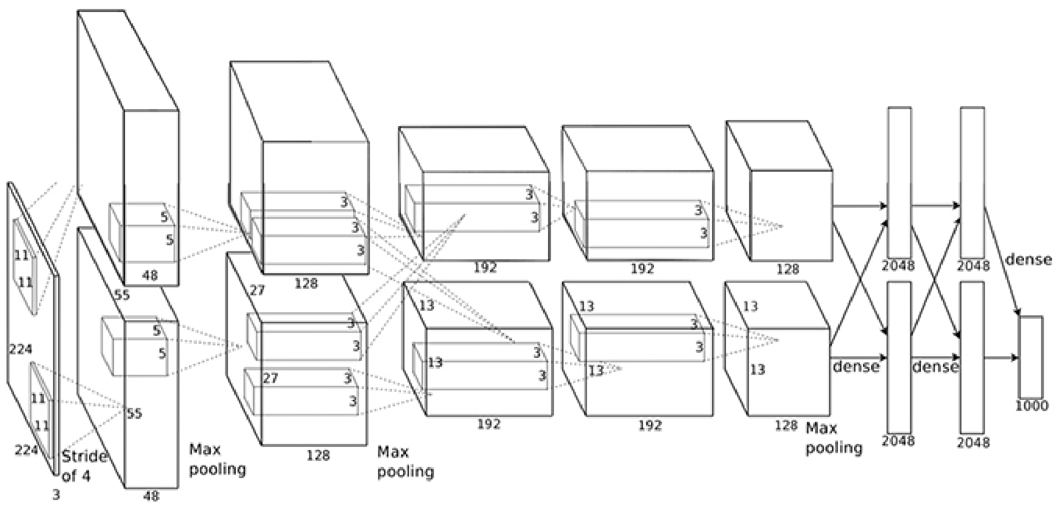


Figure 2.2: AlexNet architecture [22]

In 2014 the **VGG** architecture is introduced by Simonyan and Zisserman in their paper *Very Deep Convolutional Networks for Large Scale Image Recognition*.[32] They apply a pretraining method, which trains the smaller networks at first and then uses these results as initialization for larger networks and get a top-5 error rate of 7.30%. Despite it's advantages argue Mishkin and Matas in 2016 that all you need is a good init.[26] Instead of the pretraining method, researchers prefer Gaussian initialization, Xavier initialization[18] and MSRA initialization.[20]

Figure 2.3: VGG architecture [32]

The champion of the ILSVRC 2014 is **GooLeNet** [35] with a top-5 error rate of 6.67%. This CNN architecture is also known as "Inception module " which consists of 22 layers. The number of parameters decrease from 60 million (AlexNet) to 6.79 million.

The winner of the ILSVRC 2015 is the Residual Neural Network (ResNet) of Kaiming He et al.[20] It is worthwhile mentioning that the human top-5 classification error rate on this dataset has been reported to be 5.1%. The ResNet achieves a top-5 error rate of 3.57% which beats human. But it still has 60.2 million parameters.

Finally, Table 2.1 shows the key figures around these networks: In terms of accuracy, DL models for Image Classification have been developed a lot from 1998 to 2015. However, parameter numbers of every new model is more than 60 millions that far beyond the capacity of a laptop. As a result, LeNet-5 is a classic CNN architecture which meets the needs of this work with acceptable accuracy and lower computational cost. A number of modifications will be done to fit this architecture to the research goal.

The basic structure of CNNs are identical: feature extraction parts are composed of various CNOV and pooling layers, while classification parts are FC layers. An image passes through a series of layers to get the output which is a set of numbers with different probabilities of classes describing the image. To understand how CNN architectures work and be able to do

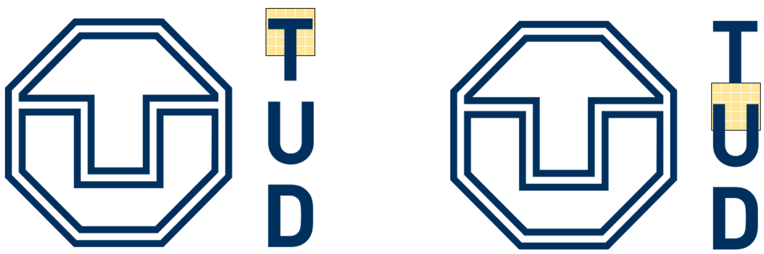
Name	LeNet-5	AlexNet	VGG	GoogLeNet	ResNet
Year	1998	2012	2014	2014	2015
Top-5 Error	-	15.30%	7.30%	6.67%	3.57%
Data Augmentation	-	+	+	+	+
Number of Convolutional Layers	3	5	16	21	151
Layer Number	7	8	19	22	152
Parameter Number	6.E+04	6.E+07	1.E+08	7.E+06	6.E+07

Table 2.1: Comparing between different CNN architecture

modifications, basic information about input layer, convolutional layer (CONV), Activation Layer, Pooling Layer and fully-connected (FC) layer as well as how they work together should be interpreted.

2.2 Convolutional Layer

The term **convolution** refers to an orderly mathematical procedure, in which two sources of information are intertwined and a new information is produced. In the case of a CNN, the convolution is used on the input data with the kernels (filters) to produce feature maps.



(a) Feature matched (b) Feature didn't match

Figure 2.4: Visualization of the filter on the image

Supposing that there is a flashlight that is shining from top left of the input image (see Figure 2.4). This flashlight shine covers a 5×5 area. The flashlight first slides across the image horizontally, then comes down and slides in the next row horizontally, until the entire area of this image is covered once. That is how convolution in machine learning works. This flashlight is professionally called a kernel or a filter, and the region which the kernel shines over is named as receptive field. The kernel is a matrix with values which are called weights or parameters in machine learning. Those values will be put in use second time when the weight matrix moves along the image. This basically enables parameter sharing in a CNN. The stride is the number of pixels, with which we slide our filter horizontally or vertically.

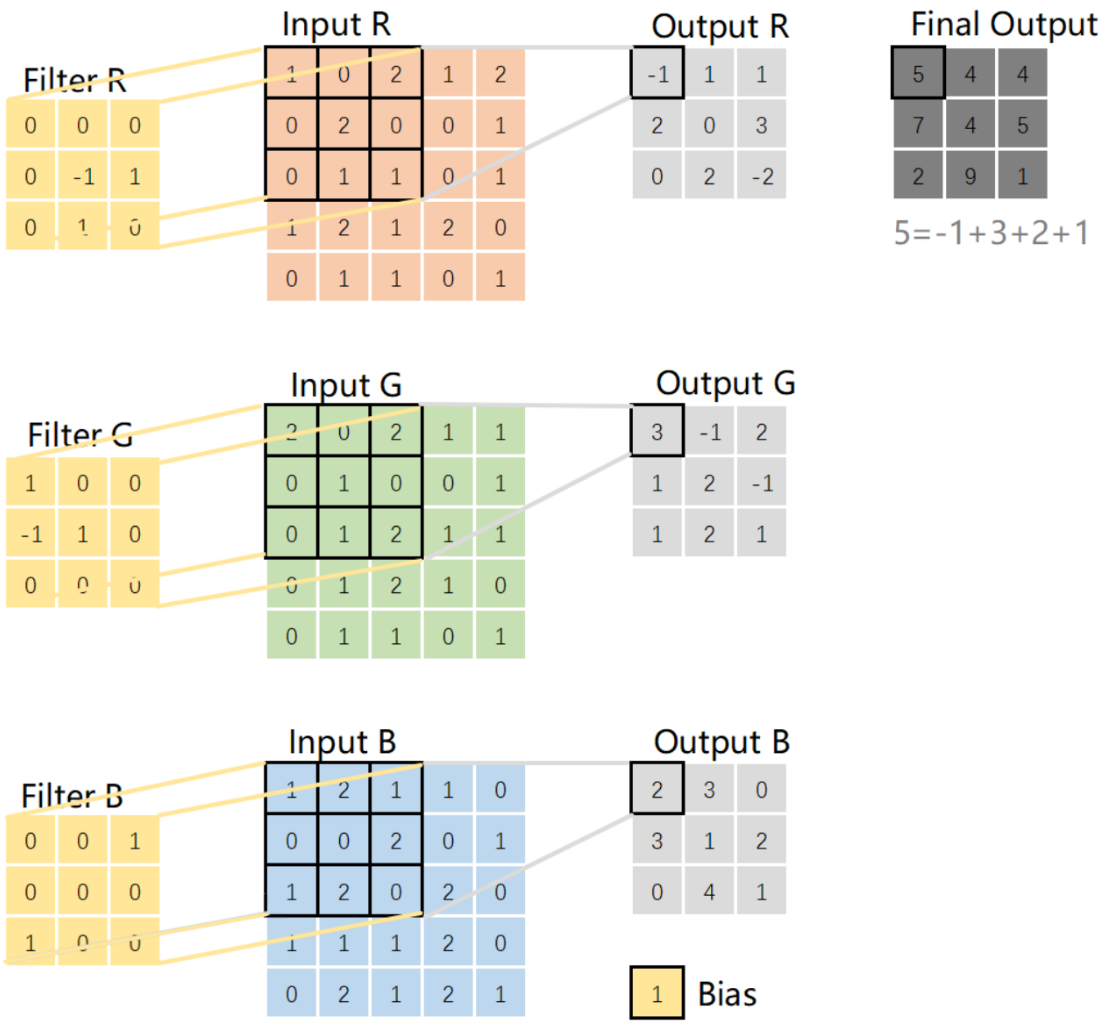


Figure 2.5: How to convoluting

As illustrated in the Figure 2.5, the filters are now sliding over the respective input image channels and producing a processed feature map of each. The convolution happens between the values in the filter and the original pixel values of the image. Out of mathematical reasons, the depth of this kernel should be exactly the same as the depth of the input, so that the dimension of this kernel is $3@3 \times 3$. After the convolution we get a $3@3 \times 3$ feature map out from the $3@5 \times 5$ input. Some kernels may have stronger weight than others to give more emphasis to certain input channels than others. In real case, each of the three channel processed output is then summed together to form one channel. Lastly, each output filter has one bias term. The final output will be bias plus the previous output.

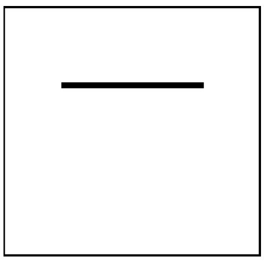
The roll convolution, which actually plays here from a high level, is feature identifiers. The features are some of the simplest characteristics that all images have in common. A line would be a good example. As shown in Figure 2.6 (a) and (b), the filter is 5×5 and is going to be a line detector (assuming that the input is a black and white picture). As a line detector, the filter has a pixel structure. In which there are higher values along the curve shaped area.

We can now convolute the multiplications between the parameters of the filter and the original pixel values of the input image.

Feature matched

What will happen if there is a shape from the input image that generally matches the curve this filter is representing? The summation of multiplication will result in a large value as illustrated by Figure 2.6 here.

$$(1 \times 60) + (1 \times 60) + (1 \times 60) = 180$$



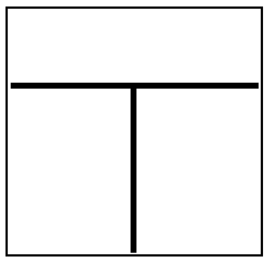
(a) Visualization of a Line Detector Filter

0	0	0	0	0
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

(b) Pixel Representation of Filter

0	0	0	0	0
60	60	60	60	60
0	0	60	0	0
0	0	60	0	0
0	0	60	0	0

(c) Pixel Representation of Receptive Field



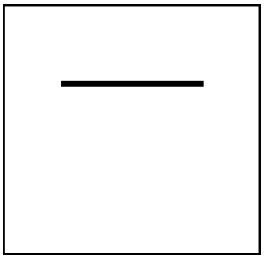
(d) Visualization of Receptive Field

Figure 2.6: Feature matched

Feature didn't match

On the contrary, if we move our filter and there is nothing in the input image that responds to the curve detector filter, the result of the multiplications summed together will be a much lower value as figure 2.7 shown below.

$$(1 \times 0) + (1 \times 0) + (1 \times 0) = 0 \tag{2.1}$$



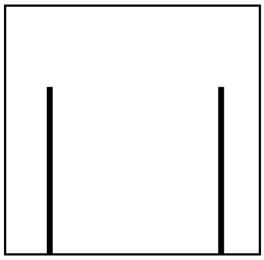
(a) Visualization of a Line Detector Filter

0	0	0	0	0
0	1	1	1	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

(b) Pixel Representation of Filter

0	0	0	0	0
0	0	0	0	0
60	60	0	60	60
60	60	0	60	60
60	60	0	60	60

(c) Pixel Representation of Receptive Field



(d) Visualization of Receptive Field

Figure 2.7: Feature did not match

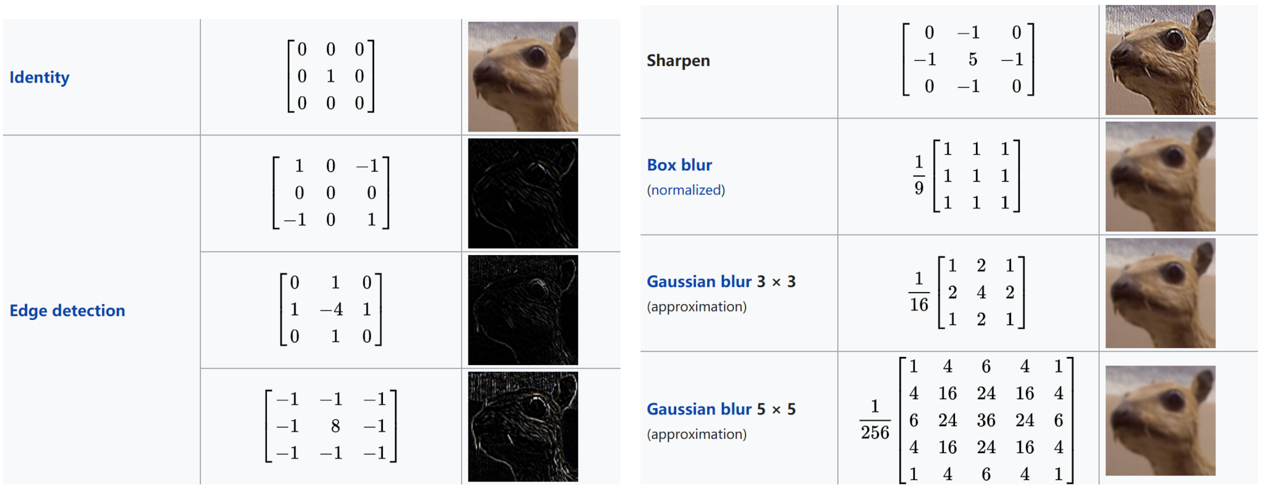
The small example is a visualized filter that only detects line. We can have different filters to detect various kinds of features, for example, curves to the left, curves to the right and straight edges.[17]

Feature engineering

We always want something from the input images which contain a lot of unnecessary information. In order to achieve that, we convoluting images with kernels to filter the distracting information out. Jannek Thomas applies a Sobel edge detector (similar to the kernel above) on the input data to filter the outlines and shapes out of the image. For this reason, the application of convolution is often called filtering, and the kernels are often named filters.

The procedure of taking inputs, transforming them and feeding the transformed inputs to an algorithm is called feature engineering. There are dozens of different kernels with varying functions, for instance those that sharpen or blur the image, and each feature map may help our algorithm to carry out better on its task. Feature engineering is so painful, because for each type of data and each type of problem different features are required.

However, when we extract information from images, is it possible to automatically find the most suitable kernels for a task?



(a) (b)

Figure 2.8: Different kernels [9]

Feature learning

Convolutional nets do exactly this. Rather than having fixed numbers in the kernel, we first assign random parameters to these kernels which is trained on the data. As the training process of the CNN goes on, the kernel does better and better at filtering a given image (or a given feature map) for relevant information. This process is automatic and is called feature learning. Without such difficulties mentioned before by feature engineering, feature learning automatically generalizes filters to each task. We simply need to train the network to find the best filters. This is what makes convolutional nets so powerful.

2.3 Pooling Layer

After setting the convolution layer, it is common to insert a pooling layer between successive convolutional layers. Just like convolution, pooling operates on each image (feature map) yet without filters. Instead of computing the sum of the multiplication, pooling computes the average of the pixel values in each window (average pooling) or takes the maximum pixel values and abandon the rest (Max Pooling).

Max pooling is the most common form of pooling that is applied to the pooling layer. As you can see in the example, both the specified stride and the pooling size are 2. The operation is applied to each depth dimension of the convoluted output (feature map). As illustrated in the figure below, the 4×4 feature map becomes 2×2 after the max pooling operation.

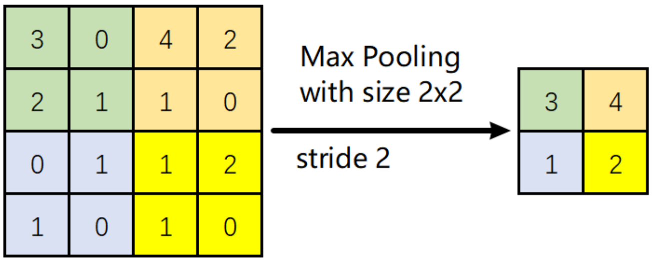


Figure 2.9: Max pooling

The following example shows how max pooling looks on a real image. In this example, the input volume of size $[224 \times 224 \times 64]$ is pooled with filter size 2 and stride 2 into output volume of size $[112 \times 112 \times 64]$.

(a) Origional image (b) After max pooling

Figure 2.10: Max pooled image

As shown in Figure 2.10 The max pooled image still retains the information, while the dimensions of the image is halved.

There are many reasons for the proven effectiveness of the idea of pooling in practice. Pooling can reduce the feature map size as the layers get deeper while at the same time keep the significant information. It helps to reduce the number of parameters and memory consumption in the network. This also shortens the training time and controls overfitting. Moreover, pooling provides basic invariance to rotations and translations, and improves the object detection capability of convolutional networks. The larger the size of the pooling window, the more in-

Pooling in some sense tries to do feature selection by reducing the dimension of the input. Overfitting can be simply thought of as fitting patterns that do not exist due to the high number of features or the low number of training examples. So by selecting a subset a subset of features, you are less likely to find false patterns.

formation is condensed, which leads to slim networks that fit more easily into GPU memory. However, if the pooling size is too large, it will cause a predictive performance decrease because too much information is thrown away.

2.4 Active Function

Human brain learns things by firing electrical impulse from one neuron to another in the hierarchy. In the programming world, researchers simulate biological electrical impulse in neutral networks with activation functions. The primary goal of these functions is to convert an input signal to an output signal.

In an ANN, a neuron sums all their products of inputs (X) and their corresponding weights (W). Following this product is an addition of bias. Before the output is transmitted to the next neuron, an activation function $f(x)$ would be used to decide whether this value should be delivered as an input to the next neuron or even better to be a zero (not activated). For example:

$$Y = Sigmoid(W \cdot X + Bias) \tag{2.2}$$

Moreover, activation functions can also introduce Nonlinear properties to the Network. Without activation function, an ANN is a linear regression model, because the output signal always is a linear function. A linear function can be defined as a polynomial with the highest exponent equals to one. In contrast, a nonlinear function is a function which is not linear and has a curvature when they are plotted.

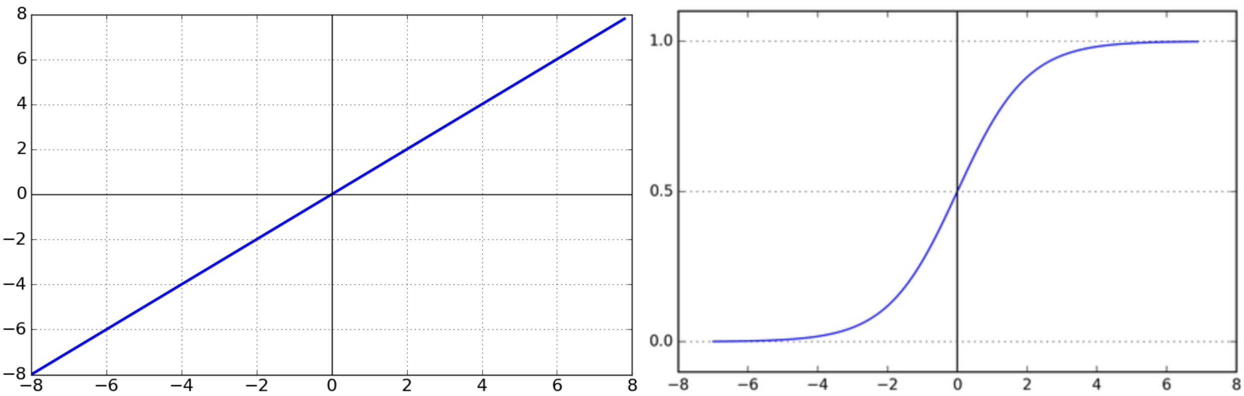
ANN can be considered as a Universal Function Approximators, which means no matter what function we want to realize, there is an ANN model with appropriate parameters that can accomplish the mission. To process more complicated data inputs such as images, audios, and videos, it is necessary to make ANN more powerful. With activation functions we are able to give ANN the ability to learn complex data and generate non-linear mappings between inputs and outputs.

In backpropagation optimization strategy we compute the gradients of errors (loss) with gradient descent optimization or other techniques to reduce error. That requires a differential of the activation functions.

In the following text, four kinds of activation functions are reviewed:

- Linear or Identity activation function
- Sigmoid Function
- Hyperbolic Tangent Function (Tanh)
- Rectified Linear Unit (ReLU)

After going through all the activation functions above, the most appropriate one for this diploma thesis will be chosen.



(a) Linear activation function (b) Sigmoid function

Figure 2.11: Linear activation function and Sigmoid function

Linear or Identity Activation Function

As shown in Figure 2.11 (a), the function is linear. Consequently, the output of the functions is not limited between any range.

Equation:

$$f(x) = x$$

Linear Function helps nothing with the complexity or various parameters of usual data which is fed to the ANN. Instead, the most frequently used activation functions are the nonlinear activation functions.

Sigmoid or Logistic Activation Function

The Sigmoid Function has a characteristic ”S”-shaped curve. As shown in Figure 2.11 (b), Y values are steep between the X values from -2 to 2. The output of this function is confined between 0 and 1.

Equation:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Since the probability of anything exists only from 0 to 1, though Sigmoid function can be used as a classifier considering its property, there are also problems of this function: at both ends of the Sigmoid function Y values tend to change very little in response to the change of X, which means the gradient in this range is very small. This leads to a problem of ”vanishing gradients”. This can cause an ANN to get stuck at the training time. The logistic Sigmoid function can cause a neural network to get stuck at the training time.

As a more generalization of the Sigmoid function, the Softmax function works not only for binary classification problems but also for multi-class classification problems.

Tanh or Hyperbolic Tangent Activation Function

Tanh function has a similar ”S” - shaped curve, as Sigmoid function, but its’ range is from -1 to 1.

Equation:

$$f(x) = tanh(x) = \frac{1}{1 + e^{-2x}} - 1$$

In fact, it could be seen as a scaled sigmoid function:

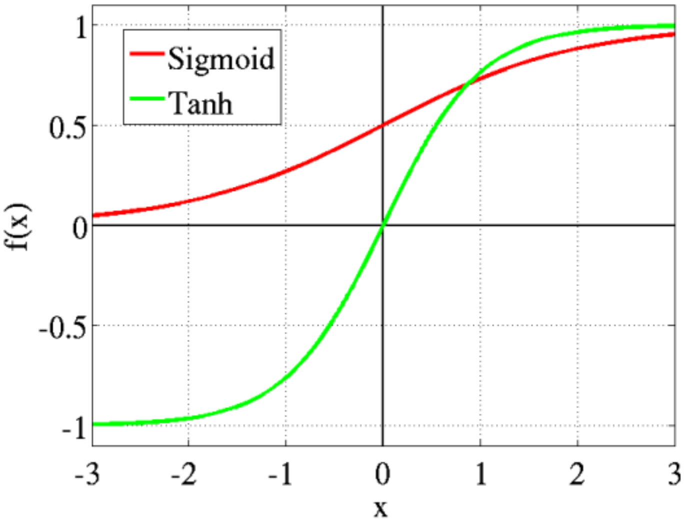


Figure 2.12: Tanh and logistic sigmoid

$$f(x) = 2 \cdot Sigmoid(2x) - 1$$

The advantage of Tanh function is that with negative, zero or positive inputs you will also get negative, zero or positive mapped value respectively. It makes Tanh function suitable for the two classes classification tasks. However, for the similar reason of Sigmoid function, Tahn function still suffers from the vanishing gradient problem.

ReLU Activation Function

The Rectified Linear Unit (ReLU) activation function has become a very popular activation function in DL currently. Since it is used in almost all the CNNs or DL. The range of ReLU is $[0, \infty)$, which means that only a non-negative x-value yields and outputs.

Equation:

$$R(x) = max(0, x)$$

There are many advantage of ReLU activation function: The mathematical form of this function is very simple and efficient. With a network using randomly initialized weights, approximately half of the neurons have 0 as output because of the mentioned characteristic of ReLU. That is to say, the network is very light. In Machine Learning field, the best techniques and methods are the most simple and consistent. As Avinash Sharma V says: "ReLU is less computationally expensive than Tanh and Sigmoid because it involves simpler mathematical operations.".

Another advantage of this function is that it avoids the vanishing gradient.

However, ReLU should only be used in the hidden layers of ANN but not in the output layer. For the classification in the output layer we need to calculate the probabilities for each class, which Softmax function is capable of.

Another problem with ReLU is that any input with negative values yields zero. With the horizontal line for negative X in ReLU, the gradient towards 0. It means the weights not get updated during the training. This weakens the power of the ANN to fit or train what from the input data suitably. As a matter of fact, any input with negative values fed to the ReLU activation function will output the value zero. This problem makes the model not fit the negative values properly and can cause several neurons to die.

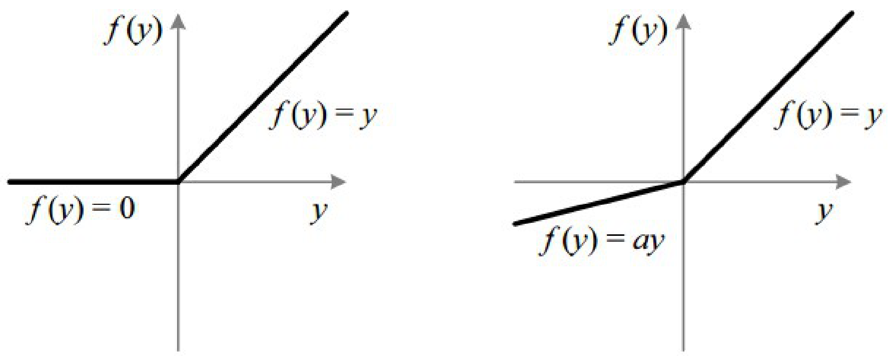


Figure 2.13: ReLU and Leak ReLU

To fix the "negative X" problem, **Leaky ReLU** function is used. By introducing the horizontal line with a slight slope, the gradient is not zero and the neurons can keep the updates alive.

Conclusion

Due to the vanishing Gradient Problem, Sigmoid function and Tanh function is not applied in the DL model. Instead, ReLU should be used in the hidden layers. Leaky ReLU function should be used when the model suffers from dead neurons problem.

In this Diploma thesis the inputs are only the images with a positive value. Thus, ReLU activation function is the best choice for the hidden layer. To get the classification job down, a Softmax function should be applied in the output layer.

2.5 Fully Connected Layer

The whole classification network can be divided into two main parts: feature extraction part and classification part. The convolutional layers serves the purpose of feature extraction, while the Fully Connected (FC) layers classify data into various classes. To make the model end-to-end trainable, we need to figure out a non-linear function to connect those extracted high-level features of the input image. In CNN, this non-linear function is learned by a set of FC layers, which aims to map the extracted features into a class probability distribution. As can be seen from Figure 2.14, in a FC layer, every neuron is connected with all the neurons in the previous layer. FC layers in CNN are identical to a fully connected multilayer perception (MLP) structure. With suitable weight parameters, FC layers could create a stochastic likelihood representation. After combining relevant high-level features, the Dresden Frauenkirche was found in the input image.

If the last layer is a FC layer:

$$y_i^{(l)} = f(z_i^{(l)}) \quad \text{with} \quad z_i^{(l)} = \sum_{j=1}^{m_{(1)}^{(l-1)}} w_{i,j}^{(l)} y_i^{(l-1)} + b_j \tag{2.3}$$

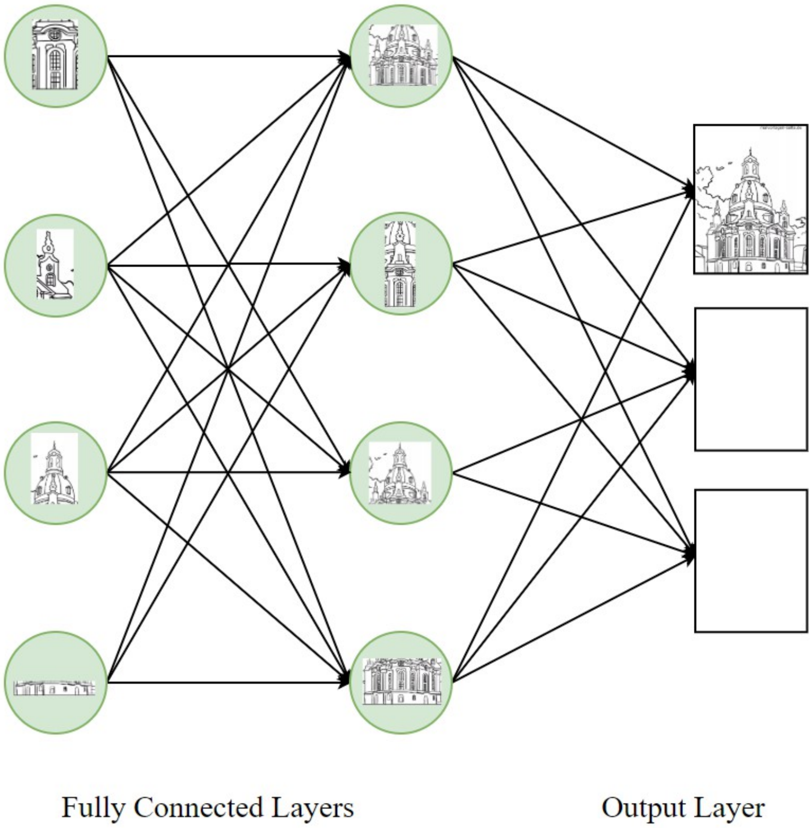


Figure 2.14: Visualization fully connected layers

If the last layer is a convolutional layer:

$$y_i^{(l)} = f(z_i^{(l)}) \quad \text{with} \quad z_i^{(l)} = \sum_{j=1}^{m_1^{(l-1)}} \sum_{r=1}^{m_2^{(l-1)}} \sum_{s=1}^{m_3^{(l-1)}} w_{i,j,r,s}^{(l)} y_i^{(l-1)} + b_j \tag{2.4}$$

where $y_i^{(l)}$ is the output of FC layer, f is active function. The input z_i of FC layer equals to the summation of dot product between weights $w_{i,j}$ or $w_{i,j,r,s}$ and output of previous layers $y_i^{(l-1)}$ plus bias b_j. FC layers output an N-dimensional vector, where N is the number of classes that the model need to identify. In this work, N would be 2 because two different classes need to be identified at the same model. For example, the FC layer of classification model gets an output $[0.15, 0.85]$, which represents that the probability of the input being an image with label 1 is 15%, and the probability of an image labeled with 2 is 85%. The labels are settled before training, for example label 1 stands for image with a crack and label 2 is a Joint. Briefly, FC layer connects with high level features that extracted from convolutional layer with particular weights, and outputs the probabilities of different classes.

2.6 Transfer Learning

The transfer learning method can apply the weights of an already trained DL model to a different but related problem. After modifications and adjustments to the network for the new task, a transferred network can be employed to train new models. Most DL models use the transfer learning approach because this method can save a lot of time from training a completely new feature extractor.

Because of the shortage of data in this new task related to natural stones, we can use the transfer learning method to address this issue. Instead of starting the training process from the very beginning, this method starts with features that have been learned from an old task, where a lot of labeled training data are available.

Transfer learning is meaningful when a new task and an old task have the same input. For example, they both use audios or images as input. Besides, transfer learning is usually used if the old task has more data than the new task. Additionally, the low-level features in the old task could be helpful for learning a new task.

In Neural Networks, the models usually detect edges in their earlier layers, shapes in their middle layers and some task-specific features in the later layers. The transfer learning methods use the early and middle layers and only re-train the later layers.

Chapter 3

Stone Defects Description

Cracks and deformation, detachment, features induced by material loss, discoloration and deposit, and biological colonization are the general defects as stone materials aging. In Germany, France and all over the world, cultural heritages are destroyed because of stone aging. For example, sugaring in Figure 3.1 (a) on the head of a marble sculpture is found in Munich, Germany. Limestone element of a cathedral in France has peeled off (see Figure 3.1 (b)). And there is a network of thin cracks on the sculpture in Versailles, France.

(a) Sugaring (b) Peeling (c) Thin cracks

Figure 3.1: Cultural heritages with stone defects [6]

We classify the quality grade of the stone products by studying the existence or the size of these defects. The durability of natural stone can be affected by different factors, for instance, poor structure, incorrect bedding, lime run-off and frost attack or acid rain. In the table below you can see more information about natural stone defects with delicate classification.

Damage	**Details**	
Crack and Deformation	Crack	Fracture, Star crack, Hair crack, Craquele, Splitting
	Deformation	
Detachment	Blistering	
	Bursting	
	Delamination	Exfoliation
	Disintagration	Crumbling, Granular disintegration
	Fragmentation	Splintering, Chipping
	Peeling	
	Scaling	Flaking, Contour scaling
Features induced by material loss	Alveolization	Coving
	Erosion	Differential erosion, Loss, Rounding, Roughening
	Mechanical damage	Impact damage, Cut, Scratch, Abrasion, Keying
	Microkarst	
	Missing part	Gap
	Perforation	
	Pitting	
Discoloration and deposit	Crust	Black crust, Salt crust
	Deposit	
	Discolouration	Colouration, Bleaching, Moist area, Staining
	Efflorescence	
	Encrustation	Concretion
	Film	
	Glossy aspect	
	Graffiti	
	Patina	Iron rich patina, Oxalate patina
	Soiling	
	Subflorescence	
Biological colonization	Biological colonization	
	Alga	
	Lichen	
	Moss	
	Mould	
	Plant	

Table 3.1: Natural stone clad defects classification [6]

This work will focus on nature stone cracks due to time and data limitation.

Chapter 4

Detecting Cracks

4.1 Databank Generation

The dataset includes totally 40,000 RGB images with 227×227 pixel resolutions. All the dataset files are stored in databank folder. The whole dataset is evenly divided into two groups as positive crack and negative crack images for classification as shown in Figure 4.1. All the images are collected from various METU Campus Buildings by Lei Zhang et al (2016). To generate the dataset, a total of 245 high-resolution images with 4032×3024 pixel resolutions is cropped into 40,000 smaller images with 227×227 pixel resolutions. Data augmentation like random rotation or flipping is not used in this dataset. In order to get a robust classifier, the smaller size of input images is used to train the network, so that this network can be able to process any larger images than 227×227 pixel resolutions.

4.1.1 Assign Labels to Images and Feed Dataset into The Model

The next step after building the model is to feed it with image dataset. There are many ways to get this job done in TensorFlow. In the text below, we can see what are the differences between TFRecord format and Raw format, and then choose a proper solution for this work.

(a) Negative (b) Positive

Figure 4.1: Examples of images in the dataset

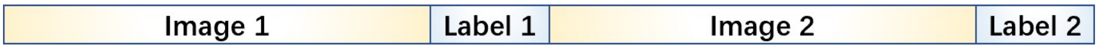


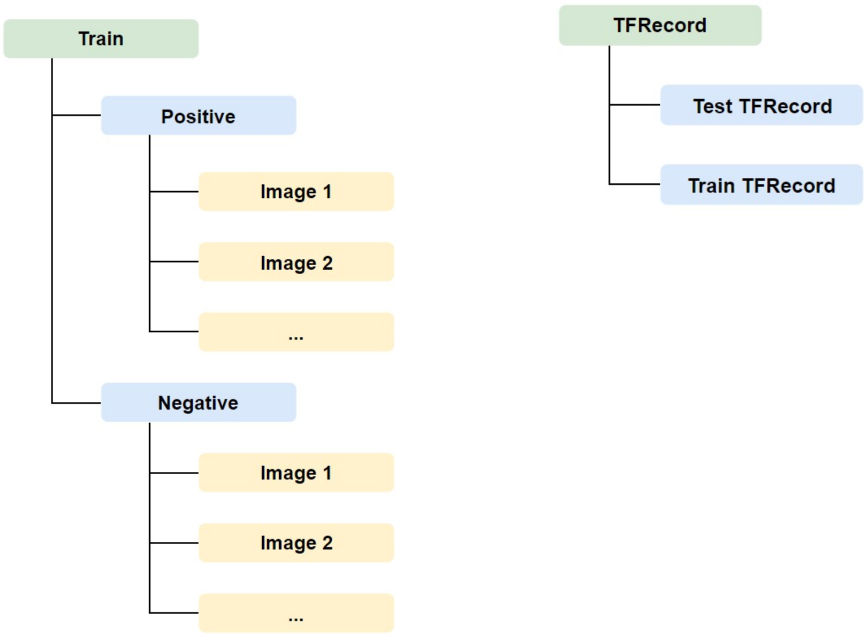
Figure 4.2: The visualization of TFRecord format

TFRecord

The visualization of TFRecord format in figure 4.2 shows a binary storage format which stores all the image information and label information as a sequence of binary strings. This format is recommended by the TensorFlow official website and described as efficient. No matter how many images are in the databank, one TFRecord file is able to store all of them and their corresponding labels.

Step 1. Specifying and organizing dataset: The first step before writing TFRecords is to specify and organize the dataset. As can be seen from Figure 4.3 (a), images with the same labels should be grouped together. The reason for that is TFRecord stores all the data as a sequence of strings, and an organized file structure helps during the reading and writing process.

Step 2. Splitting dataset and creating TFRecords: The next step is to write TFRecords and split the dataset into training and testing subsets as shown in Figure 4.3 (b). The training dataset is fed to the model in the training process. Instead of training loops, we call it training steps in machine learning. The training dataset is used again in every steps of training process so that the model learns from these data. Only after the model is completely trained, the testing dataset is then used to test and evaluate this model. Tarang Shah wrote in 2017 that the dataset split ratio is not fixed.[30] Out of the total 40,000 images, 400 of them (1%) are randomly picked



(a) Datasets structure (b) TFRecords structure

Figure 4.3: File structure

out as a testing set in this work. With the function *tf.python_io.TFRecordWriter* we are able to write data into TFRecord files as we want.

Step 3. Reading and Checking TFRecords: With the first two steps TFRecords are created. Before being put into use in the training process, it should be read first. The *tf.TFRecordReader* can read the image and label information, which is later used by *tf.train.shuffle_batch* to train the model. The shuffle function builds a shuffling batch queue that serves the purpose of preventing over-fitting from affecting the generalization ability of the model. Moreover, in algorithms like stochastic gradient descent (SGD), shuffling batch helps to ensure that we are less likely to converge to a solution lying in the global minimum for the whole training set, but more likely to find a solution that generalizes better.

Meanwhile, we can also check the TFRecords after reading it by using *tf.decode_raw* to decode image information from TFRecord and *tf.reshape(image, [227, 227, 3])* to help to define the 277 x 277 resolution of the image with RGB channels. The function *tf.cast(features['label'], tf.int32)* can read the label information of the image. If everything goes well without dataset, we can extract the picture with corresponding label by *img.save(swd+str(i)+'_' 'Label_'+str(l) +'.jpg')*. With fresh images named with their labels out of TFRecords, we can now check if labels match with images and if testing set and training set have duplicate images, and confirm whether the dataset is correctly split. Therefore, the TFRecord checking step is of great benefit to us.

Raw Image

Step 1. Splitting dataset and adding labels: The first step is to split the dataset into the training set and testing set. As by TFRecords, we use the same dataset and the same split ratio with 1%, in other words, we have 400 random images as the testing set and the rest 39600 images as the training set. Before the training process starts, we need to add labels on the training set, so that our model can learn the feature from images with the same labels. All the images in training set need to be renamed with the same *label_Number.jpg* format, for example, *Positive_19991.jpg* and *negative_00035.jpg*. In order to improve the efficiency of the rename operation, a batch file (see Appendix A) is employed.

Step 2. Disorder dataset: The next step is to disorder the dataset, in order to prevent overfitting and make the model converge faster than using SGD algorithm (see TensorFlow Step 3 for more details). Before random ordering the dataset, we need to save all images and labels in a temp file: *temp = np.array([image_list,label_list])*, and then use *temp = temp.transpose()* to transport the array. Finally, a numpy function is used to finish the random operation *np.random.shuffle(temp)*.

Step 3. Reading dataset: In the last step, we save all the key information into a temp file and transposed it. So the first column of the temp file refers to image information, and the second column is about label information. With *image_list = list(temp[:,0])* *label_list = list(temp[:,1])* *label_list = [int(i) for i in label_list]* we can read those information from the temp file. To make the image information readable for TensorFlow, they should at first be converted to string and then decoded by TensorFlow: *image = tf.cast(image,tf.string)*, *image = tf.image.decode_jpeg*, *(image_contents,channels =3)*. We set the channels as 3, because the dataset consists of color images. Similarly, label information should be converted to int32 format.

It is worthwhile mentioning that the value of images should be standardized to make gradient descent converge more quickly in two aspects.

First of all, we can speed up gradient descent by feature scaling, because gradient descends quickly on small ranges and slowly on large ranges. Standardization (or Z-score normalization) is the process of re-scaling features in statistics, so that the multiple features can take on similar scale (range of values). The idea is to make sure that all the values from different features can have a Gaussian distribution with $\mu = 0$ and $\sigma = 1$, where μ is the mean and σ is the standard deviation of all values in image:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2} \tag{4.1}$$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{4.2}$$

with $\sigma = 1$ the gradient descends more quickly than on the original range. However, the values of each pixel in RGB channel already have a similar scale (from 0 to 225), we hence apply an adjusted standard deviation where N_{RGB} calculates the number of pixels in each of the three channels:

$$\sigma_{adjusted} = \frac{max(\sigma, 1)}{N_{RGB}} \tag{4.3}$$

So the standard scores (or z scores) are calculated as follows:

$$Z = \frac{x_i - \mu}{\sigma_{adjusted}} \tag{4.4}$$

Secondly, standardized data is distributed around the center of 0 ($\mu = 0$), which makes convergence of the gradient more effective. If the input data are all positive, then the output value of ReLU activation function of each layer is also positive (see Figure 2.13). This ends up in the back propagation with all positive or negative gradient of the weight w (because the output values of activation function depended on the the error that passed to the current layer). As a result, the optimal path of gradient descent becomes a zigzag path, which is undesired for a effective convergence gradient descent algorithm.[1]

Standardization process can be operated by a TensorFlow inbuilt function:

`tf.image.per_image_standardization` .

4.1.2 Comparison of Two Different Formats

Classification ability of the model is not affected by the method of feeding images, since both datasets used the same images. After labeling operation, datasets are fed to the same model for training. However, there are still differences between the two methods. As shown in Table 4.1, TFRecords can be read very fast, because it stores all the data as a sequence of strings. But before reading, TFRecords needs to be created, which consumes a lot of time. On the other hand, TFRecords requires about 10 times more storage space. In contrast, raw images consumes less time and less storage resources. As a result, raw images format is used in this work for datasets.

	Images	TFRecords
Time of Creating Dataset/s	0	380
Time of Reading Dataset/s	170	40
Total Time assumption	170	420
Space of Storage/MB	234	1874

Table 4.1: Resource consumption of different format

4.2 From Classic CNN to Crack Detector

4.2.1 Layer Pattern of Classic Network Structures

As previously noted in section 2.1, CNNs are commonly made up of three types of layers: the convolutional layer, pooling layer and fully connected layer. We focus on the layer pattern in this section and build a CNN architecture for crack detection in the next section.

As can be seen from the previously mentioned CNN structures, most structures have a similar pattern: CNN uses CONV layers to extract features. Activation functions are always applied in the next step to add nonlinear. After that a pooling layer is used to reduce the feature map size as well as the number of parameters. Finally, FC layers work as a classifier to give us the result of the predictions. This is supported by the study from Stanford Vision and Learning Lab, as they described this CNN pattern with:

$$INPUT \rightarrow [(CONV \rightarrow RELU) \times N \rightarrow POOL] \times M \rightarrow [FC \rightarrow RELU] \times K \rightarrow FC \tag{4.5}$$

where CONV stands for convolutional layer, $\times$ means a reasonable repetition of this layer, and POOL indicates the possibility to apply a pooling layer with $0 \leq N \leq 3$, $0 \leq M, 0 \leq K \leq 3$.

Besides, a stack of small size convolutional layers is more efficient than one large size convolutional layer for two reasons: The first is that this kind of small size CONV layers stack has fewer parameter, coinciding with that obtained by studies from Szegedy et al, which benefits the training time.[36] The second reason is out of expression ability of nonlinear. Since a stack of small size CONV layers have obviously more activations than a single CONV layer.[38]

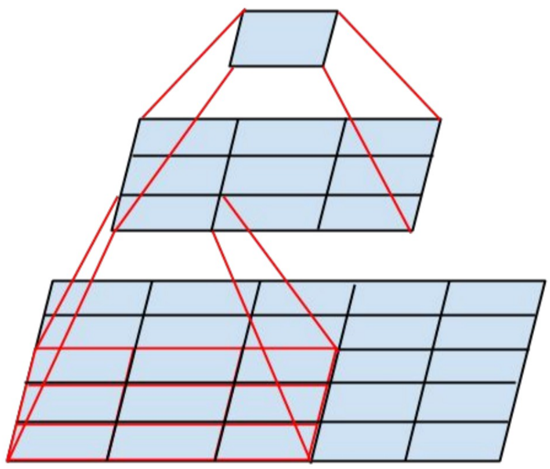


Figure 4.4: Replacing the 5×5 CONV layer with two 3×3 CONV layer [38]

An example of it is the comparison between a stack of two 3×3 CONV layers and another of 5×5 CONV layers (see Figure 4.4): A neuron on the first CONV layer has a 3×3 view of input volume. On the second layer, each neuron has a 3×3 view of the first CONV layer as well as a 5×5 view of the input volume. Supposing this is an image with C channels with input_channel

= output_channel, then the two 3×3 CONV layers need $2\times3\times3\times C\times C = 18\times C^2$ parameters, while one 5×5 contains $5\times5\times C\times C = 25\times C^2$ parameters. Compared with two 3×3 filters, a single filter contains $25/18 = 1.39$ times more parameters. As a result, sliding this small 3×3 CONV set over the input image is more reasonable than using a 5×5 CONV layer, because a stack of small filters expresses more features of the input image with less parameters.

4.2.2 Network Structure for Crack Detection

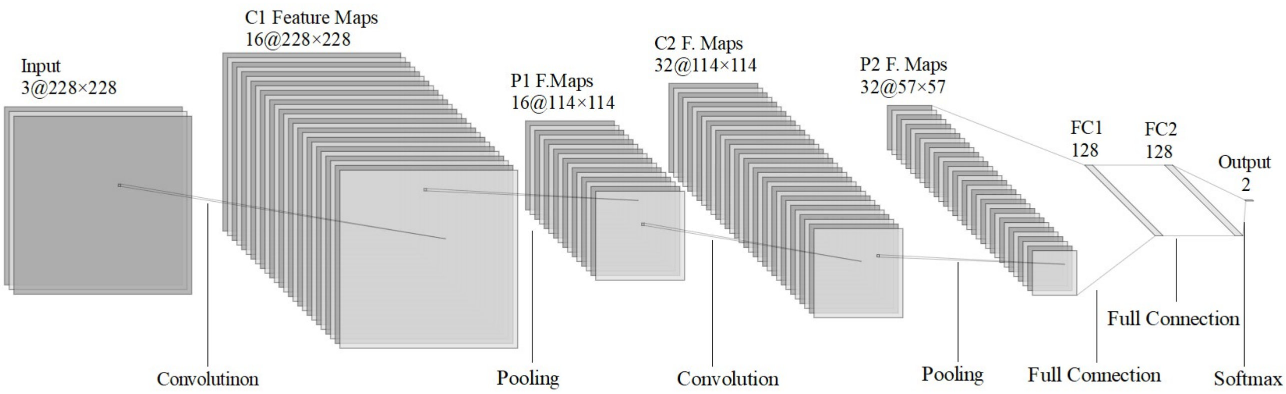


Figure 4.5: Network structure for crack detection

After the introduction of the classic CNN architecture in previous text, this section builds the architecture of our crack detector. According to the Table 2.1, LeNet is the best structure for this work, because it meets the need of computational cost and has an acceptable accuracy. In the building process of our new network architecture, changes and modifications are necessary to let this architecture fit to the research goal. As shown in Figure 4.5, CONV layers are labeled C_x, pooling layers are labeled P_x and FC layers, where x is the index of layer. All calculations are described in the later section.

The first layer is an input layer, where the original LeNet and the modified version in this work differ from each other: Instead of one channel black and white images, the three channels colors are fed into the network. The input images are re-sized to 228×228 pixels, in case some images in the dataset are with different sizes.

The second layer of this image classification network is a CNOV layer with 16 feature maps.

Every unit of feature map is connected to a 3×3 area in the input image. Both strides in horizontal and vertical are 1, which means every time the filter moves to 1 pixel on the input. During the CNOV operation, a SAME padding technique is used. Layer C1 contains 448 trainable parameters and 824,464 connections.

As aforementioned in section 4.2.1, a single 5×5 CONV layer can be replaced by a stack of two 3×3 CONV layers, which has fewer parameters with the same input and output size. There are two main downsides for CONV operation: First of all, the output volume shrinks (see Figure 2.5. Secondly, the pixels in the corner are scanned only once while the pixels in the middle get covered more than once. As a matter of fact, we extract more information from the middle pixels. Therefore, zeros are commonly used to pad the border in practice. In TensorFlow we can chose between VALID Padding and SAME Padding. VALID Padding means no padding and every right-most column or bottom-most row are dropped, if the image width is not a multiple of the filter width or the image height is not a multiple of the filter height. In contrast, SAME Padding uses zeros to pad around the image to make sure the output size is the same as the input size after CONV operation. As can be seen from Figure 4.6, the size of the input volume doesn't change after CONV with SAME padding.

Additionally, in propose of introducing nonlinear properties to the Network, CONV layers are always followed by an activation function. The original LeNet uses Sigmoid as activation function, which suffers from vanishing gradient problem. As discussed in the preceding section, ReLU is accordingly the best choice among all the activation functions for this work, because the inputs images are always positive values.

The third layer is a max pooling layer that contains 16 feature maps of size 114×114. Each unit in feature map is connected to a 3×3 neighborhood in the feature map of CONV1 layer. Both strides in horizontal and vertical are 2. In spite of 207936 connections, layer P1 has zero parameter, since it has no weights and biases.

原本是average pooling

Pooling operation makes the number of rows and columns half as the feature maps in CONV 1. It helps to reduce the parameter number and memory consumption. Compared with other pooling methods, max pooling is capable to keep features on the feature map.

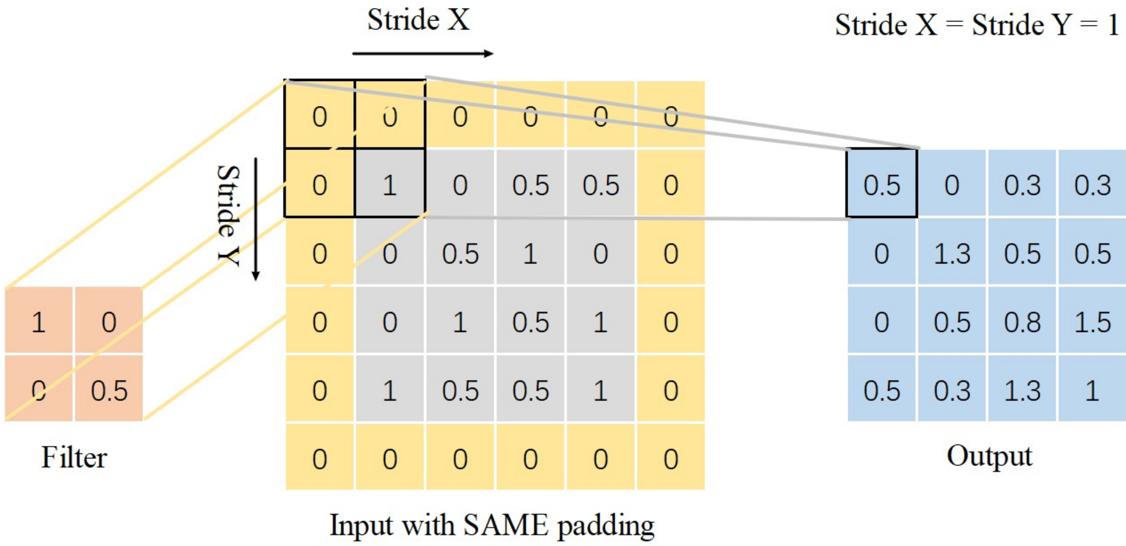


Figure 4.6: SAME padding

Inspired by AlexNet, a Local Response Normalization (LRN) method is employed here to imitate real neurons, which creating competition for big activities among neuron outputs computed using different kernels.[22] Details of LRN can be found in the next subsection.

The fourth layer is a CNOV layer with 32 feature maps of size 114×114. Every unit of the feature map is connected to a 3×3 area in the input image. According to Szegedy etl.: *Higher dimensional representations are easier to process locally within a network. Increasing the activations per tile in a convolutional network allows for more disentangled features. The resulting networks will train faster.*[36] That is to say, the feature map number of CONV layer should get more to help disentangling features, which makes a faster training. Both strides in horizontal and vertical are 1. During the CNOV operation, a SAME padding technique is used. Layer C2 has 4640 trainable parameters and 415,872 connections.

The fifth layer is again a max pooling layer containing 32 feature maps of size 57×57. Each unit of the feature map is connected to a 3×3 neighborhood in the feature map of CONV1 layer. Both strides in horizontal and vertical are 2. Layer P2 has zero parameter, since it has no weights and biases, but 51,984 connections. It should be pointed out that the feature maps

Layer	Input Volum			Filter Size			Stride		Output Volum			Parameters
	Di	Hi	Wi	K	Fx	Fy	Sx	Sy	Do	Ho	Wo	
Input	3	228	228	-	-	-	-	-	3	228	228	0
C1	3	228	228	16	3	3	1	1	16	228	228	448
P1	16	228	228	-	3	3	2	2	16	114	114	0
C2	16	114	114	32	3	3	1	1	32	114	114	4640
P2	32	114	114	-	3	3	2	2	32	57	57	0
FC1	32.	57	57	1	-	128	-	-	1	128	-	13308032
FC2	1	128	-	1	-	128	-	-	1	2	-	258

Table 4.2: Filter shape and parameter number of each layer

of P2 is now 32@57 × 57, but the input of FC layers need to be a 1st-order tensor. Similar with 1 × 1 CONV operation in LeNet, this converting operation will be done by a TensorFlow inbuilt function *tf.reshape(P2,[-1])* .

The sixth layer is a FC layer with 128 units. Each unit connects with all units in the previous layer, which makes F1 has an amount of 13,308,032 trainable parameters. A ReLU activation is processed on the outputs of F1 layer.

The seventh layer is still a FC layer with 128 units. F2 layer is fully connected with F1 layer. There are 258 trainable parameters in this layer. A multiFC layer set composed of F1 and F2 gives the network a stronger expression ability nonlinear to connect those extracted features from previous layers. A ReLU activation is again processed on the outputs of F2 layer. The output of F2 is an arbitrary real value vector with the shape $1 \times N$ with $N = 2$, since the classes N for this work is 2. Softmax function takes this N-dimensional vector of arbitrary real values and produces another N-dimensional vector with real values in the range $(0, 1)$ that add up to 1.0, e.g.: The two element vector such as $[2, 3]$ gets transformed into $[0.269, 0.731]$. The calculation of this function can be described with

$$S_j = \frac{e^{a_j}}{\sigma_{k=1}^{T} e^{a_k}} \tag{4.6}$$

where S_j is the softmax value of element j, a_j is the original value of the element, and T is the number of elements in the vector.

The Table above walks through the filter shape, the layer shape and the parameters of each layer. Following formulas describe those calculations in details. There is a volume with size $W\_i \times H\_i \times D_i$ as input to this CONV layer, and in this layer there are K filters with size of $F_x \times F_y$ sliding with stride S_x and S_y in x and y directions.

For CONV layers: the output size with SAME Padding is $W_o \times H_o \times D_o$, where [12]

$$W_o = W_i/S_x \tag{4.7}$$

$$H_o = W_o/S_y \tag{4.8}$$

$$D_o = K \tag{4.9}$$

The number of zeros need to be padded on height Z_h and on width Z_w that are calculated by:

$$Z_w = (W_o - 1) \times S_x + F_x - W_i \tag{4.10}$$

$$Z_h = (H_o - 1) \times S_y + F_y - H_i \tag{4.11}$$

Thus, $Z_w/2$ zeros are padded on the top as well as on the bottom of this image.

As parameters are shared in CONV layer, this layer introduces $F_x \times F_y \times D_i$ weights per filter and K biases: [24]

$$ParameterNumber = (F_x \times F_y \times D_i + 1) \times K \tag{4.12}$$

E.g.: Layer C1 has totally $(3 \times 3 \times 3 + 1) \times 16 = 448$ parameters.

For pooling layers: The output size is $W_o \times H_o \times D_o$, where [24]

$$W_o = (W_i - F_x)/S_x \tag{4.13}$$

$$H_o = (W_o)/F_y \tag{4.14}$$

$$D_o = K \tag{4.15}$$

Pooling layers introduce zero parameter since there is no weights and biases in this operation.

For FC layers: The number of weights is calculated with: $W_i \times H_i \times D_i \times F_y$. Especially F_y is here the number of neurons. The number of biases is the same with neuron numbers F_y. In Conclusion,

$$ParameterNumber = (W_i \times H_i \times D_i \times F_y) \times K + F_y \tag{4.16}$$

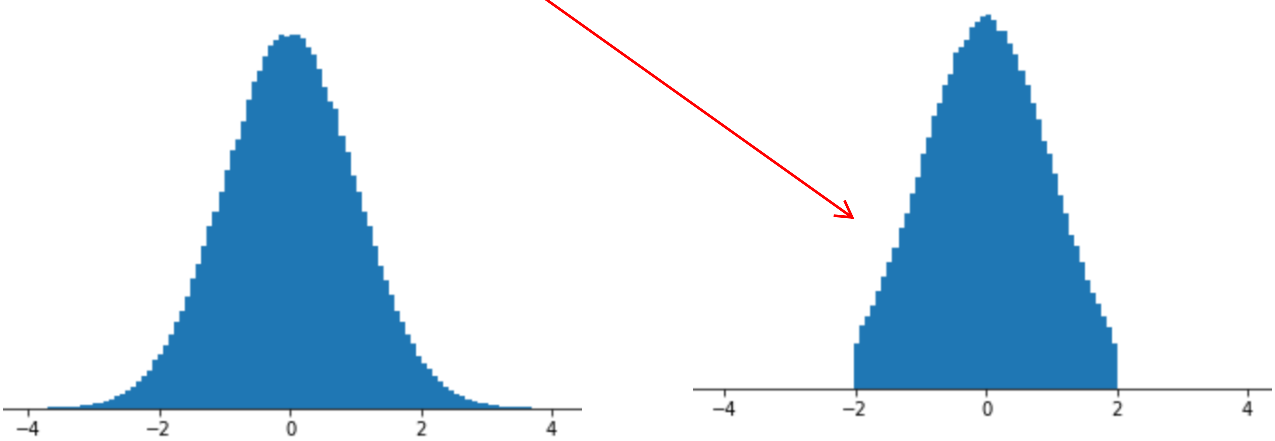
E.g.: Layer F1 has totally $(57 \times 57 \times 32 \times 128) + 128 = 133,080,32$ Parameters.

4.3 Hyperparameters Tuning

In machine learning, hyperparameters are the variables set before a training process begins. They determine how the network is structured as well as how the network is trained. Complete details of these hyperparameters will be provided in this section.

The value of num_threads controls the maximum number of threads enqueuing 'tensors', in other words, it controls the number of multiple threads that prepare training examples and push them in the queue. This value is dependent on the hardware. The image loading process of this network is run on a machine with Intel i78550U CPU which has 4 core and 8 threads[8]. As a result, we will run the training process on GPU with *num_threads = 8* .

Hyperparameters also includes the size of filters and the number of filters that we mentioned in the last section. Incidentally, bias and weights need to be defined, too. The shape of biases of CONV layers is the same with the feature map numbers. Thus, CONV1 has 16 biases and CONV2 has 32 biases. Meanwhile, every unit in FC layers has a bias so that both FC1 and FC2 have 128 biases. The values of all the biases are initialized before the training process as 0.1 with *initializer=tf.constant_initializer(0.1))* , which ensures that all ReLU units fire in the beginning, and therefore obtain and propagate some gradient. The shape of weights needs to be preset as well. For CONV layers weights are set with *shape=[a, b, c, d]*, where a and b are the size of filters with a = Fx; b, c is the number of input depth, and d is the number of feature maps in this layer. Thus, CONV 1 has the weights of shape [3, 3, 3, 16] and CONV2 has the weights of shape [3, 3, 16, 32]. The value of all the weights are initialized before the training process as 0.1 with *initializer=tf.truncated_normal_initializer(stddev=0.1, dtype=tf.float32))*. This makes all the weights random values with 0.1 standard deviation and follow a normal distribution with a specified mean and standard deviation. Except that values, whose magnitude is more than 2 standard deviations from the mean, are dropped and re-picked, as shown in Figure 4.7. According to Tensorflow website, this is the recommended initializer for neural network weights.[11]



(a) Standard deviation

(b) Truncated standard deviation

Figure 4.7: Truncated standard deviation

As mentioned before, LRN was applied after pooling layers. This normalization can be described with:

$$b^i_{x,y} = \frac{a^i_{x,y}}{(k + \alpha \sum_{j=max(0,\frac{i-n}{2})}^{j=min(N-1,\frac{i+n}{2})} {a^j_{x,y}}^2)^\beta} \tag{4.17}$$

$b^i_{x,y}$ is the regularized output for kernel i at position x, y; $a^i_{x,y}$ is the source output of kernel i applied at position x, y; N is the depth radius; K is base; α is a scale factor and β is an exponent. The constants k, n, α and β are hyperparameters whose values are determined using a validation set, $n = 4$, $k = 1$, $\alpha = 10^{-4}$, and $\beta = 0.75$.

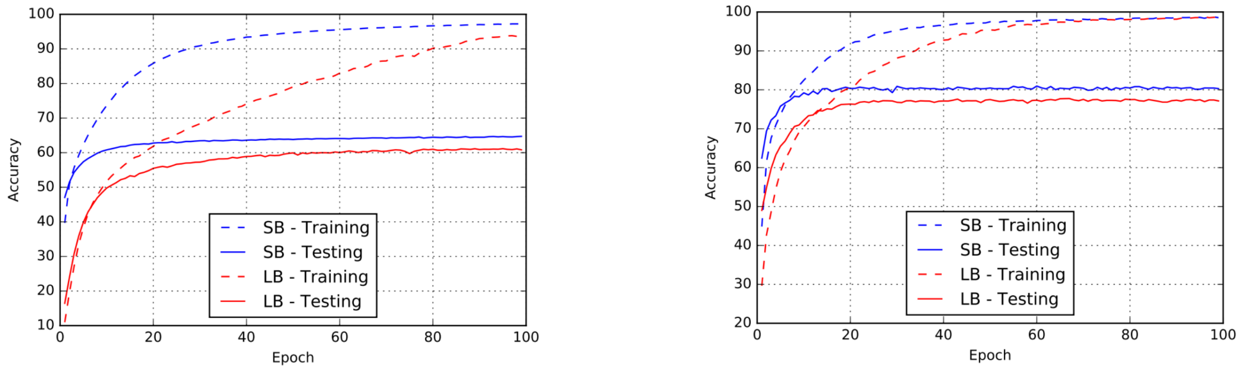
In case input images have different sizes, a resize operation with *tf.image.esize_image_with_crop_or* *_padding(image,IMG_W, IMG_H)* is needed before training. This operation resizes any image in this work to a target width *IMG_W = 228* and height *IMG_H = 228* by either centrally cropping the image or padding it evenly with zeros.

The learning rate is one of the most important hyperparameters for neuron networks. With this hyperparameter, the backpropagation process is able to know how far to move the weights in the gradient decent direction for a mini-batch.

If the learning rate is low [10], the training process takes a lot of time, but become more reliable, because very tiny updates to the weights are made. However, if the learning rate is high, weight changes can be too big to make the loss worse, in other words, the training doesn't converge. Consequently, the value of the learning rate is set to be 0.00003. This value is not fixed all the time, because an Adaptive Moment Estimation (Adam) optimizer is applied during training. Details of Adam are provided in the next subsection.

Batch size is the number of sub samples given to the network after which parameter update happens. According to Andrew NG's opinion: the extreme situation are the gradient descents using the entire dataset. In this case training process saves time by going to the right direction to a local minimum. Another extreme situation is that batch size has only one sample, which can lead gradient towards wrong direction, but reduce the computing burden. Several studies on the other hand show that large batch has limits, too. *Large-batch methods tend to converge to sharp minimizers of the training and testing functionsand as is well known, sharp minima lead to poorer generalization. In contrast, small-batch methods consistently converge to flat minimizers, and our experiments support a commonly held view that this is due to the inherent noise in the gradient estimation* [21]. Figure 4.8 shows that training and testing accuracy for small batch size is better than large batch size methods as a function of epochs. This is supported by Masters and Luschi: performance has been consistently obtained for mini-batch sizes between $m = 2$ and $m = 32$, which contrasts with recent work advocating the use of mini-batch sizes in the thousands [25]. We hence set the batch value in this work with *BATCH_ BATCH_SIZE = 32*.

Capacity is an integer, which stands for the maximum number of elements in the queue, and



(a) Model 1

(b) Model 2

Figure 4.8: Comparison between small batch training and large batch training [21]

can be three times bigger than batch size. In this work, *CAPAVITY = 256* . The relationships between batch size, queue and capacity will be presented in figure 6.11.

4.4 Training Process

In this stage, the model is gradually optimized until it meets our research goal. The model's loss plays an important roll in the training stage as well as in the evaluation stage later. This loss value reflects how far the predictions are from the right label, in other words, how bad the model is performing. Our intention is to minimize this loss value near 0 and get the softmax score as close as possible to 1.00.[3]

In LeCun's LeNet5 model, a Mean-Squared Loss (MSL) is used to calculate the loss scores. However, MSL is reported to have limitations: the MSE loss makes sense when the predictions are truer. However, if the predictions are false, the MSE loss may not be the best bet.[37]

We hence use the most widely accepted loss function: With *sparse_softmax_cross_entropy_with* textsl _logits() the sparse softmax cross entropy between the model's class probability predictions and the desired label can be calculated, and returns the average loss across the examples. This loss function could be described with:

$$L = \log S_j \tag{4.18}$$

where L is the loss value and S_j is the softmax score of prediction j.

To minimize the loss function, an optimizer needs to be applied. In TensorFlow, there are many in-build loss functions. Stochastic Gradient Decent is the simplest gradient decent method:

$$W = W - \eta \frac{\partial L}{\partial W} \tag{4.19}$$

where W is the weight, L is the loss value, and η is the learning rate. This method needs a long time to converge and can lead a divergent behavior to the loss function. Moreover, this optimizer has no ability to update the learning rate. Momentum optimizer is able to some degree control the divergent of loss function, but it may lead to a wrong direction to the gradient decent.

Unlike all the other optimizers, Adaptive Moment Estimation (Adam) is a method that computes adaptive learning rates for every parameter. Just like momentum, Adam can store an exponentially decaying average of the past squared gradients v_t, and keep an exponentially decaying average of the past gradients m_t at the same time. In fact, momentum behaves like a ball running down a slope, while Adam is more like a heavy ball with friction. The decaying averages of m_t and v_t can be calculated with:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{4.20}$$

$$v_t = \beta_2 m_{t-1} + (1 - \beta_2) g_t^2 \tag{4.21}$$

The first and second moment are estimated by m_t and v_t respectively, thus, this method is named Adaptive Moment Estimation. The authors of Adam observe that m_t and v_t are biased to zero, specifically during the initial time steps and the decay rates are small. Therefor, these biases by computing bias-corrected first and second moment estimates are counteracted with:

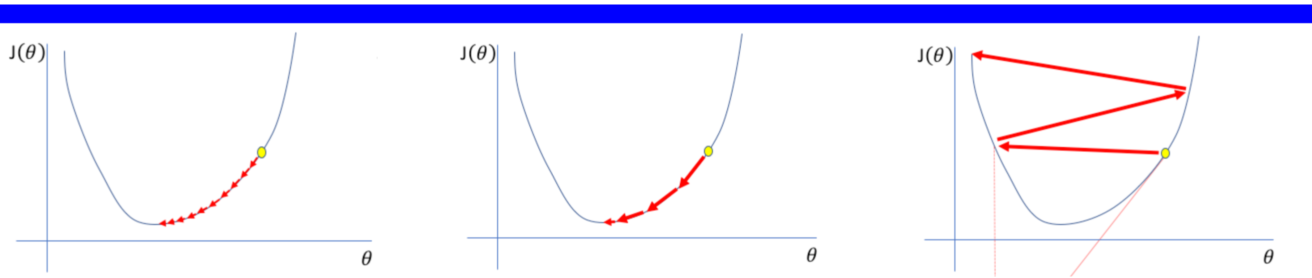
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{4.22}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{4.23}$$

where the default value is 0.9 for β_1, 0.999 for β_2 and 10^{-8} for ϵ. The parameters can then be updated with:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{4.24}$$

As for learning rate, training should start from a relatively large learning rate so that those random initialized weights are far from optimal. After using optimizer, learning rate decreases during training process to give the weight a more fine-grained updates possibility.[34] Then we can start to find learning rate that minimizes $J(\theta)$.



(a) Low learning rate (b) Optimal learning rate (c) High learning rate

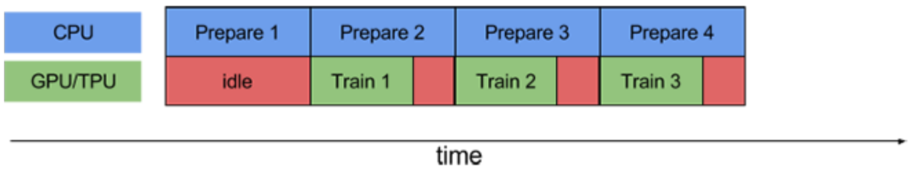
Figure 4.9: Learning rate and cost function [10]

As shown in figure 4.9 (a), a small learning rate requires many updates before reaching the minimum point. The optimal learning rate swiftly reaches the minimum point (see figure 4.9 (b)). In contrast, a too high learning rate causes drastic updates which leads to divergent behaviors (see figure 4.9 (c)). As a result, learning rate with value 0.001 which has the shortest training time is chosen.

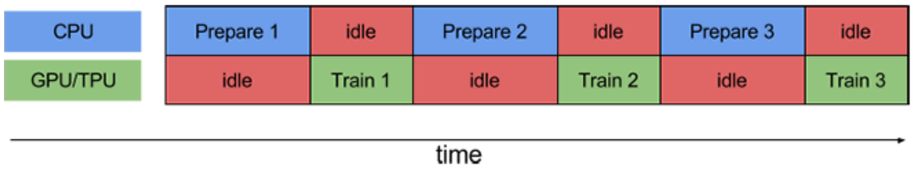
It should be mentioned that a coordinator *coord = tf.train.Coordinator()* for CPU and GPU is applied to make the training process more efficient in physical level. As can be seen from Figure 4.10, coordinator helps to reduce idleness of CPU and GPU by making them working at the same time.

Now we can finally start the training loop (see Figure 4.11):

Step 1 Feed the images and labels into the mode.



(a) Dataset with coordinator



(b) Dataset without coordinator

Figure 4.10: Coordinator [4]

Step 2 Iterate over each example in the training dataset within an step by grabbing its features and label.

Step 3 Compare the prediction of inputs and compare it with the real label. Measure the softmax value of the prediction and use that to calculate the model's loss and gradients.

Step 4 Update the model's variables with Adam optimizer.

Step 5 Repeat for each epoch.

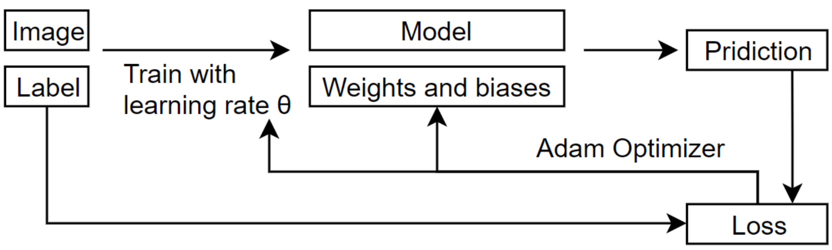


Figure 4.11: Training process

In this work, all parameters are saved as a model for every 50 steps so that we can always chose a proper model before it get over-trained. To different with other models in this work, we name this model concrete crack recognition model. Training results and evaluation of this model can be found in Chapter 6.

4.5 From Concrete to Nature Stone

With the method of transfer learning we are able to pass the training knowledge from a pre-trained model to a new model, which saves a lot of time. Our goal is to transfer the knowledge from a concrete crack recognition model to a nature stone crack recognition model.

Before training starts, we should check and read the previous model with

```
ckpt = tf.train.get_checkpoint_state(logs_train_dir)
if ckpt and ckpt.model_checkpoint_path:
    saver.restore (sess,ckpt.model_checkpoint_path)
```

Besides, a new database is made to train the model for the new propose. All the images of nature stone damages are taken by Dr. Sebastian Fuchs in the distance of 1.5 m and 3 m with RGB channels and 2560×1920 pixel of resolution, from which 75 smaller images with different contains stone crack are cropped out. By different resolutions of images in training set we can introduce noises in to the data, and that makes the model more robust. As a result, this dataset including 150 small images with different resolutions. The whole dataset is divided into a test set of 10 images and a train set of 140 images. In the training set, 70 images are labeled with crack and other 70 images are labeled with negative. As is shown in figure 4.12, different types of cracks are contained in the dataset, such as hair crack and splitting.

Next step is to start training loop. Neither hyperparameters nor network structure of previous model are modified because we have similar classification problem and same number of classes as the model of concrete detection. This time, our goal is to achieve at least a 97% train accuracy by means of transfer learning. The training processes are run on an external GeForce GTX 1070. Maximum training step is set to be 50,000. Results and evaluation of nature stone crack recognition model can be found in Chapter 6.

(a) Hair crack (b) Splitting

Figure 4.12: Examples of crack in data set of nature stone

Chapter 5

Differentiate Joint from Crack

In this chapter, a new model that can tell joints and cracks is trained. At first we make a new database which contains both cracks and joints of nature stone. Then we start to train this model with a transfer learning.

5.1 Databank Generation

For the propose of detecting joint and crack, a new database is made. This database has a number of 150 images with different pixel of resolutions. The whole dataset is divided into testing set and training set which contains 10 images and 140 images separately. The training set is composed of 70 images nature stone crack and 70 images with joints of nature stone. When it comes to test set, 5 images are nature stone with cracks and others are with nature stone joints. All the images related with nature stone cracks are cropped from RGB images with 2560×1920 pixel of resolution, which are taken by Dr. Sebastian Fuchs in distance of 1.5 m and 3 m. Images of The Great Wall from Chinese websites are collected to made the sets of nature stone joints. All images in training set are labeled while images in test set are not. As shown in the figure 5.1, figure 5.1 (a) is one of the pictures with nature stone joint, and figure 5.1 is an image of nature stone crack example.

(a) Joint of nature stone (b) Crack of nature stone

Figure 5.1: Examples of images in data set for joint and crack

5.2 Training

Our research goal is to have at least a 97% train accuracy by means of transfer learning. The training processes are run on an external GeForce GTX 1070. Max training step is set to be 6000 and learning rate is of 0.0001. Neither hyper-parameters nor network structure of previous model are modified because we have similar classification problem and same number of classes as before.

The first step is to read model information from previous model. Due to the advantage of more train images, the model information of concrete crack detection is transferred to the joint and crack detection model. And then, we start the training loop (see Figure) until research goal achieved. Training results and analysis can be found in Chapter 6.

Chapter 6

Results and Analysis

In this chapter, we analyze the results of the experiments and try to explain the reasons. At first, comparison between the affects of the different learning rates on training time and training loss is made. Then we see how did the losses and accuracies of the three models change during training and time consumption. Finaly, the working mechanism of TensorFlow is presented.

6.1 Optimal Learning Rate

As mentioned before, we start the training process with a large learning rate from 0.1, and found that the loss function doesn't improve and even started decreasing in the first few iterations. This means that any higher value doesn't help to train. Every time we train the model with a slightly smaller learning rate, until the loss function starts to decrease. During the research, Adam optimizer is applied.

As can be seen from Figure 6.1, 13 different learning rates from 0.00001 to 0.1 with Adam optimizer are trained. In each time, the network is trained until it achieved at least 97% train accuracy. The maximum step allowed is 2000. The experiments are run on an external GeForce GTX 1070. The dataset is a concrete crack dataset.

With learning rate of 0.1, 0.06, 0.03 and 0.01, the accuracy doesn't meet our goal (97 %), and from 0.006, training time begins to reduce until it gets the lowest point, which has a learning rate 0.001 with 312 seconds training time. After that, training time begins to rise slowly.

Figure 6.1: Learning rate and time

Detailed values of loss and all 13 different learning rate can be found in figure 6.2. From this graph we can see that all grey lines are learning rate values that either take to much time or can not reduce loss. For example, the training losses with learning rate of 0.1, 0.06, 0.03 and 0.01 reduce very slowly, which further explains why these learning rates in Figure 6.1 do not get our goal within maximum training step. On the other hand, the line with orange color of 0.01 learning rate reduces loss efficiently and achieves our goal. It's worth mentioning that all learning rate smaller than 0.01 are able to achieve a 97 % training accuracy within 2000 steps, and some of them even have a better performance on loss. Nevertheless, the value of 0.001 is set to the learning for the same training result and less training time. To have similar classification problem, all models in this work use the same value of learning rate.

6.2 Model Evaluating

There are three models for different research goals a trained in this work: One model that recognizes concrete cracks, another that knows nature stone crack, and the last one that tells joints from cracks of nature stone. This section presents and evaluates training results of each

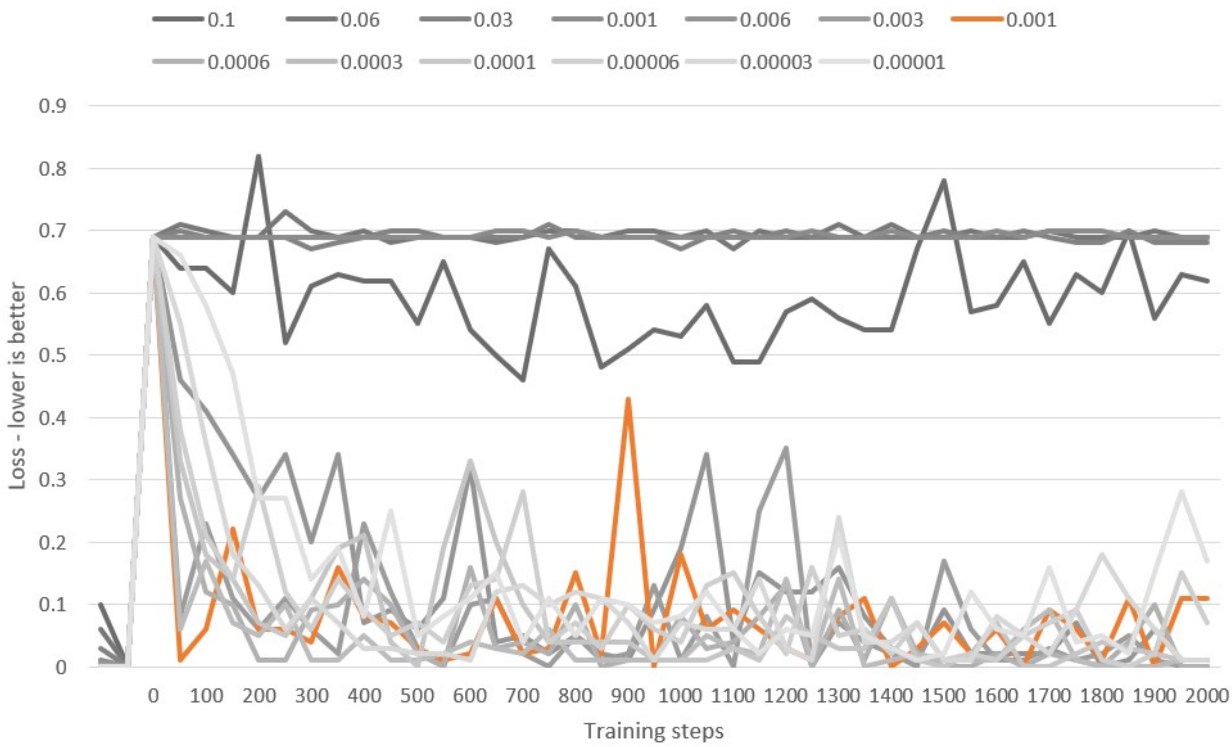


Figure 6.2: Learning rate and loss

model. After the evaluation, results for utilize are also illustrated.

6.2.1 Concrete Crack Recognition Model

Training results

During training, a dataset of concrete cracks including 40,000 images is employed. The maximum training step is set to be 15,000. With the learning rate of 0.0001, the whole training process takes around 50 minutes 38 seconds (3,038 seconds). Our goal is to train a model that can recognize concrete cracks.

Figure 6.3 shows the loss and accuracy changes during training process of the concrete crack model. As can be seen from figure 6.3 (a) , those lines in purple and orange show the fluctuation of training loss and testing loss separately. Over the steps from 0 to 10,800, the values of both lines are fluctuated but show a downward trend overall. This means the model is learning and trying to descent gradient by finding the minimum loss value. In contrast, values of both lines after 11,100 training steps stabilized near 0.01 except for slight floating. figure 6.3 (b) shows some characteristics in common with Figure 6.3 (a): With the same training step, every value of accuracy corresponds to the value of loss. They both show sharp fluctuation before 10,800 training steps and trend to become steady after 11,100 training steps. Until 15,000 training steps, the accuracy of testing gets stable around 100%. The training accuracy is calculated by:

$$\frac{N_{right}}{N_{total}} \times 100 \tag{6.1}$$

where the value of N_{right} indicates the number of images with right predicted labels in training process, and N_{total} is the total number of images in this batch. Similarly, testing accuracy could be calculated with:

$$\frac{N'_{right}}{N'_{total}} \times 100 \tag{6.2}$$

where N'_{right} indicates the number of right predicted images during test process and N'_{total} stands the number of test data. According to the figures, the concrete crack model gets the best performance. Taking the model with 14,199 training steps for example, when the test loss is near to 0, the test accuracy is near to 1. We hence keep the model with 14,199 training steps for validation and transfer learning.

Utilizing

After training and evaluating, this model can be used to detecte concrete cracks. First of all, the model information need to be loaded. Then a random image from testing set is picked out and fed into the model. During the process of utilizing, the notwork outputs classification results without updating weight. The accuracy of the single sample is calculated with:

$$accuracy = e^{-Loss} \tag{6.3}$$

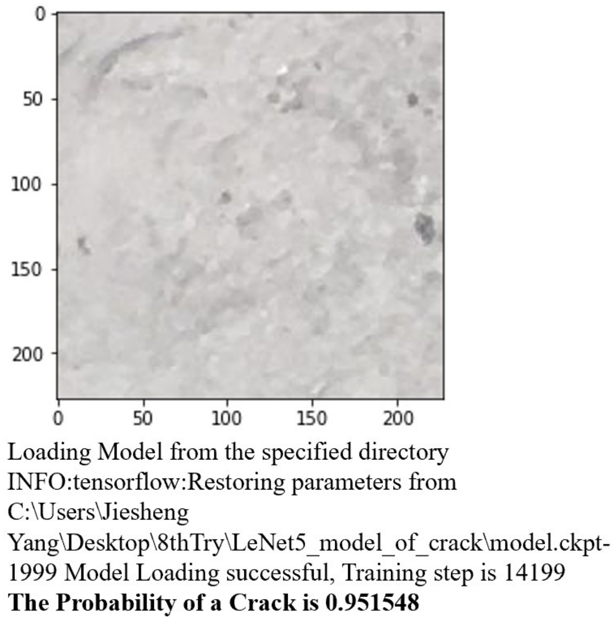
As can be seen from Figure 6.4, after loading the models of 14,199 training steps, Figure 6.4 (a) is predicted to have a crack with a probability of 0.951548. Figure 6.4 (b) is predicted

(a) Loss

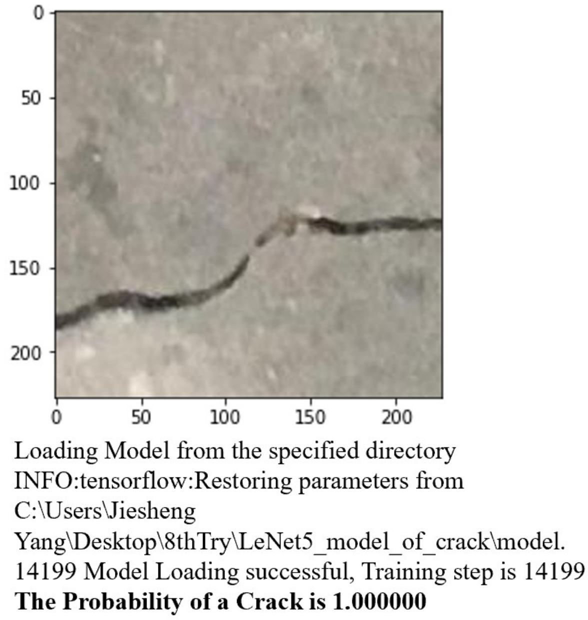
(b) Accuracy

Figure 6.3: Loss and accuracy of concrete crack detection model

the probability of no crack is 1.000000. The research goal to train a model which can detect concrete cracks is achieved.



(a) Predition results of concrete with crack



(b) Predition results of concrete without crack

Figure 6.4: Utilization of concrete crack detection model

6.2.2 Nature Stone Crack Recognition Model

Training results

The dataset for nature stone crack contains 150 images (details of this dataset can be found in section 4.5). The maximum training step is 2000. With the learning rate of 0.0001, it takes 367 seconds to finish the training process. Our goal is to train a model that can recognize nature stone cracks.

As can be seen in Figure 6.5, the red line stands for the accuracy of transfer learning, and the light purple line stands for the accuracy without using transfer learning. After first 50 training steps, the accuracy of the model with transfer learning (87.5%) is around 35% higher than the one without (53.12%). After 100 steps, however, the accuracy of model without transfer learning catches up and both methods have nothing different in terms of training accuracy. Taking time into consideration, the training with transfer learning takes 36 seconds to get a

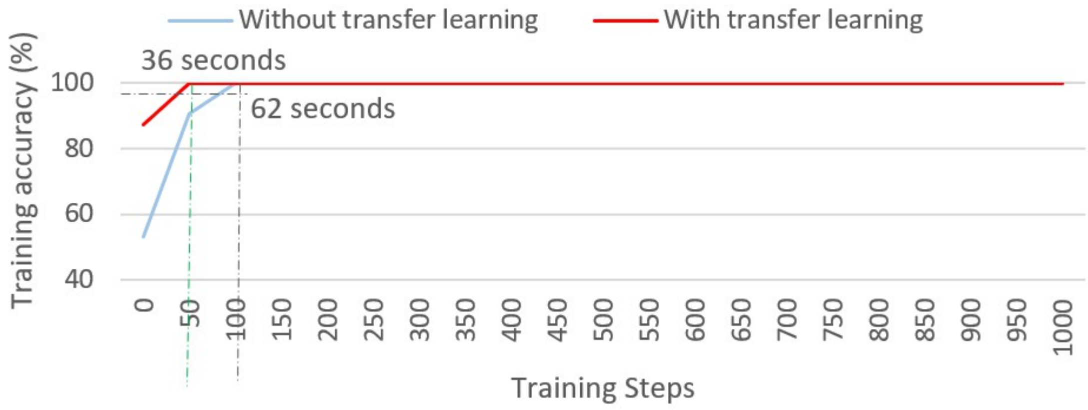
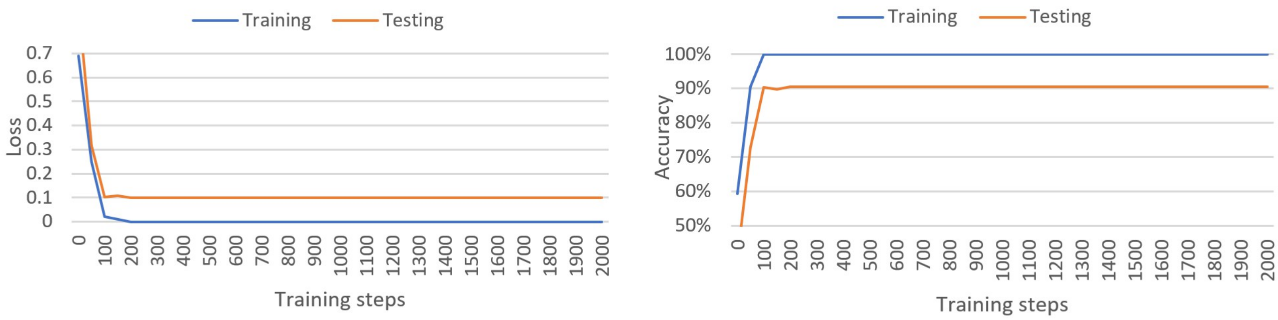


Figure 6.5: Transfer learning for nature stone crack recognition model

training accuracy over 97%, while it takes 62 seconds for the training without transfer learning to achieve the same training accuracy. The results suggest that the main advantages of transfer learning is the potential to save training time and make neural network performs better at the same training steps even without a lot of data.



(a) Loss

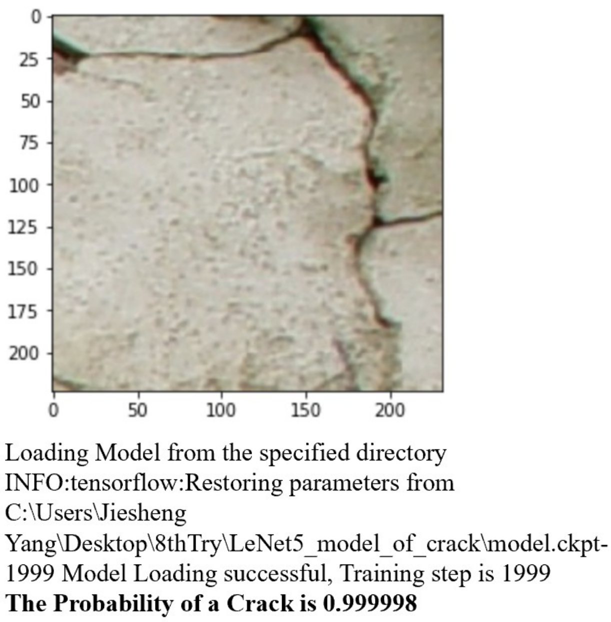
(b) Accuracy

Figure 6.6: Loss and accuracy of nature stone crack detection model

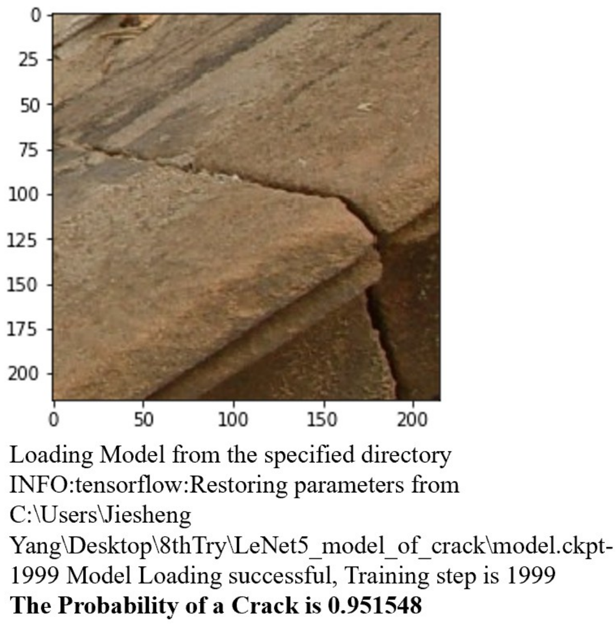
Figure 6.6 provides data regarding loss and accuracy changes while the model is training. From figure 6.6 (a), we can see that both training loss and testing loss fall sharply at the first 150 training steps. In contrast, the value of both losses from 150 steps to 2000 steps remain stable with training loss near 0 and testing loss around 0.1. Similarly, both accuracy keep steady after 150 train steps with 100 % of training accuracy and around 90 % of testing accuracy. It is clear that due to the limited number of training images, this model does not perfectly suit every image in the testing set.

Utilizing

As can be seen from Figure 6.7, after loading the models of 1999 training steps, Figure 6.4 (a) presents a nature stone that is predicted to have a 0.999998 probability with a crack on it. Figure 6.4 (b) shows the prediction results of a nature stone, which has a 0.951548 probability to have a crack. Consequently, the research goal to train model to detect cracks of nature stone is achieved.



(a) Prediction results of natural stone with crack



(b) Prediction results of natural stone with crack

Figure 6.7: Utilizing of natural stone crack detection model

6.2.3 Crack and Joint Recognition Model

Training results

The dataset for joint and nature stone crack contains 480 images (details can be found in section 5.1), and is used to train this model. The training processes are run on an external GeForce GTX 1070. The max training step is set to be 6000. With the learning rate of 0.0001, the whole training process takes 14 minutes 34 seconds (874 seconds). Our goal is to train a model that can recognize cracks and joints of nature stones.

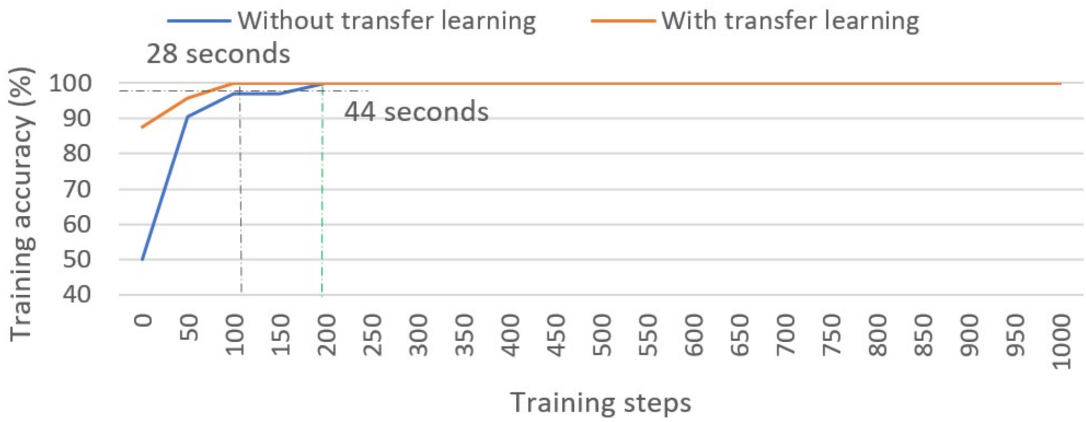
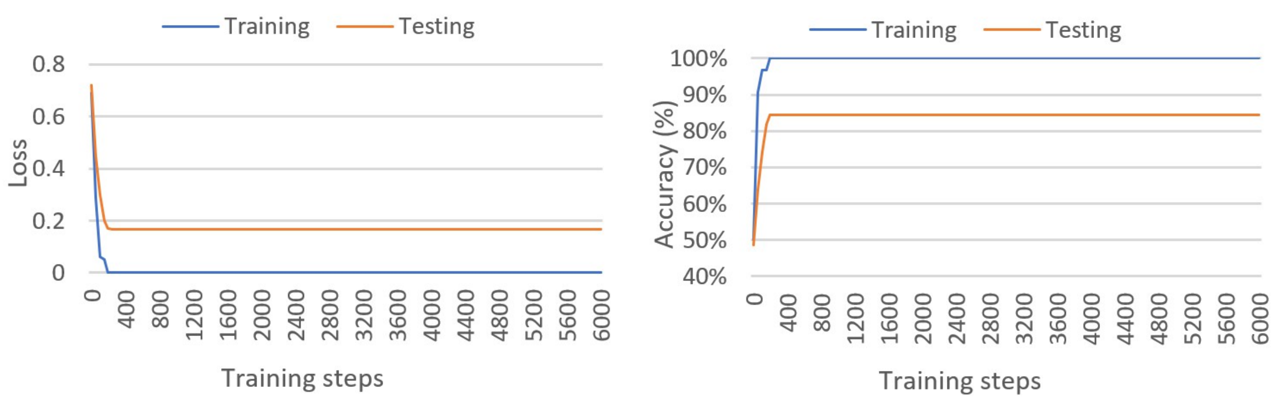


Figure 6.8: Transfer learning for crack and joint recognition model

Figure 6.8 is a line chart that illustrates the changes of training accuracy between transfer learning and normal training during training process. The figure shows that transfer learning need less time (28 seconds) than normal training (44 seconds) to get an accuracy over 97%. Besides, training accuracy of model with transfer learning has a higher value after the first training step.

Figure 6.9 shows the loss and accuracy change during training process. At the first 400 steps, training accuracy and testing accuracy increase to around 100 % and around 82 separately. In the stage from 400 steps to 6000 steps, both accuracy keep steady, and the values of training and testing loss stay stable, too. Due to the limitations of small databases, features are not learned well in this model, which leads to a relative lower test accuracy.



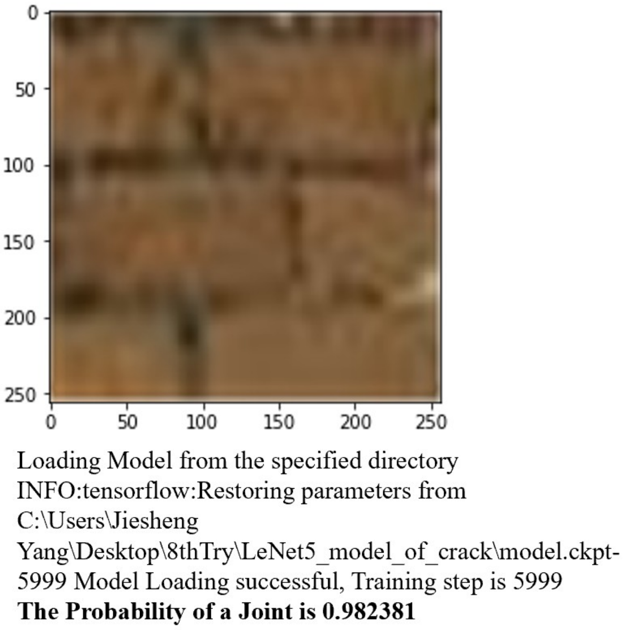
(a) Loss

(b) Accuracy

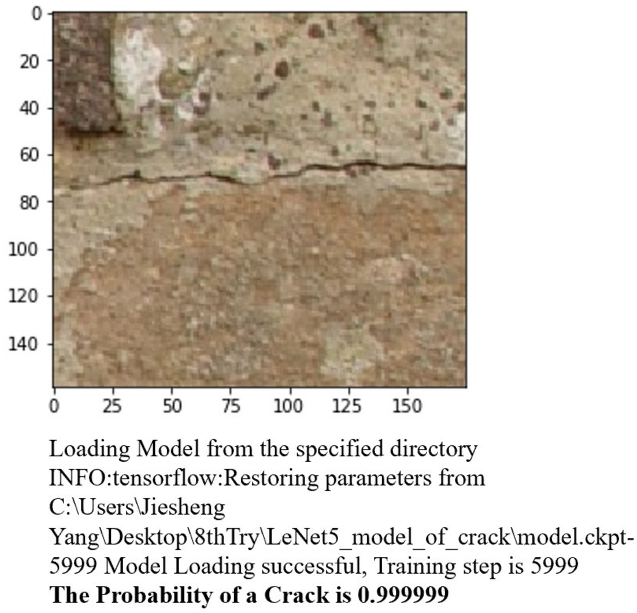
Figure 6.9: Loss and accuracy of crack and joint detection model

Utilizing

Figure 6.10 are examples of utilizing the trained model. After loading model information of 5999 training steps, Figure 6.10 (a) is predicted with a 0.982381 probability to have joints on the stone and Figure 6.10 (b) is predicted with a 0.999999 probability to have cracks on it. Thus, our research goal to train a model which can detect joints and cracks of nature stone is achieved, and this model even works with unfocused images.



(a) Prediction results of natural stone with joint



(b) Prediction results of natural stone with crack

Figure 6.10: Utilizing of natural stone crack detection model

6.3 Working Mechanism of TensorFlow

To summarize the working mechanism of TensorFlow (see Figure 6.11), performing a training step should at first use multiple threads to prepare the image data and corresponding labels, and push them in the queue. The length of this queue is defined with capacity. Every time a mini batch of BATCH_SIZE elements is picked out to train the network and calculate the loss of this batch. This loss is updated by the trainable parameters as well as the learning rate, until we have a stable loss near zero.

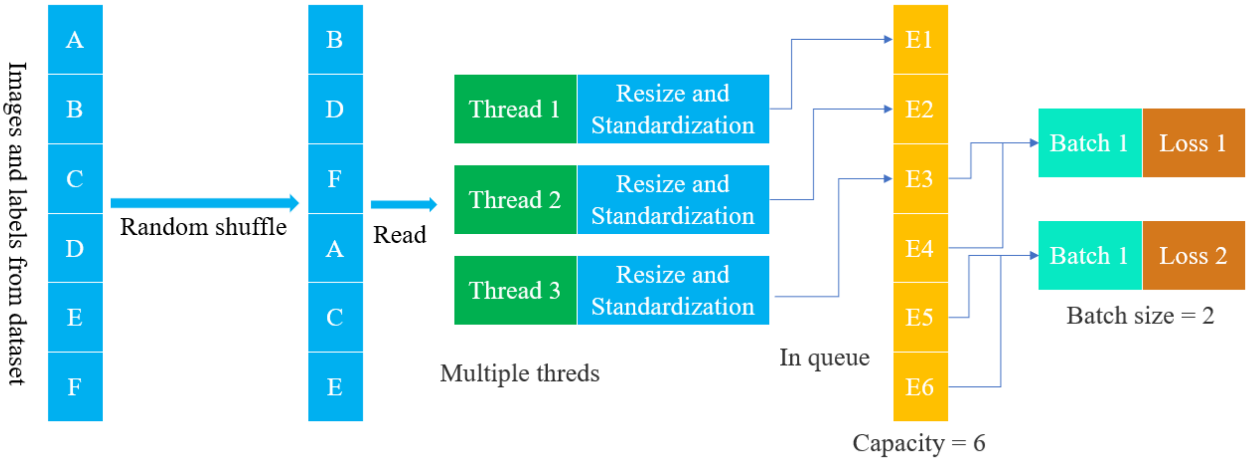


Figure 6.11: Working mechanism of TensorFlow

Chapter 7

Discussion

In this work, we have built one network with a suitable structure and two databases. Three models are trained and validated: one that recognizes concrete cracks, another that recognizes nature stone cracks, and the other that distinguishes joints and cracks of nature stones. In this chapter, we will at first make discussions and further analyze the results. Secondly, a comparison between our work and other studies will be made. Finally, limitations and potential improvements to the current work is presented.

7.1 Discussions

After comparing the training results in this work, it is clear that training loss reduces sharply in the early stage of training, and tends to become stable in the subsequent training. This suggests that we can set the max step with a slightly larger value to observe the loss curve, so that we could decide if the loss is steady. For example, in figure 6.6 (b) the loss of training barely changes after 200 steps, but in order to observe whether the model has stabilized, we still trained the model till 1800 steps.

Concerning the input format, TFRecord needs more time for its' generation and more storage space, but has a faster reading speed as shown in Table 4.1. We hence believe that TFRecord is an optimal choice if there are a large amount of samples in database and enough storage space in the hard disk. This is because TFRecords can be used repeatedly once it is generated. Moreover, with a faster reading speed, a lot of time can be saved during the tune of hyper-parameters in the repeatedly training process.

The training accuracy of the nature stone detection model and of the model that distinguishes joints from cracks are limited. According to the textbook of University Toronto [2], whether the model is over-trained can be told from the line chart of cost values. As shown in Figure 7.1, the model is under-fitting if the training loss and the test loss are both high. With over-fitting, on the other hand, the training loss is low while the test accuracy is still very high. Before the model is over-fitted, the loss of validations slopes down at first until it hits the minimum point and then starts to slope up again. The results of loss suggest that those two mentioned models are well trained, and the reason for the limited test accuracy is that the size of training set is too small for model to learn all the features.

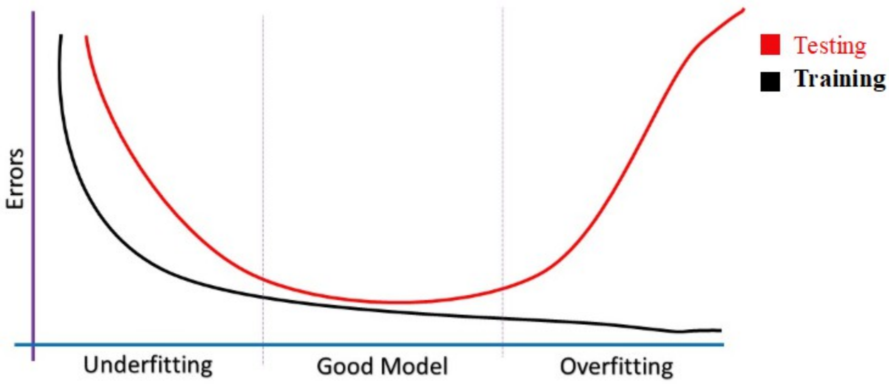


Figure 7.1: Overfitting and underfitting

7.2 Comparison

By comparing the results with those from other similar projects, we hope to determine whether our results match the state of the art methods and what new light we caste on the research of building renovation. There are three similar projects that can be compared with.

The first project was built on GiitHub by Satyen [28]. This model achieved a 85% accuracy on the test set of his own data set with a total of 8 layers. The sizes of the three CONV layers are $24@10 \times 10$, $48@7 \times 7$ and $96@11 \times 11$.

The second project was led by Dr. Cha [15]. This model achieved a 97.95% test accuracy on his own data set with a total of 6 layers. The sizes of the four CONV layers are $24@20 \times 20$, , $48@15 \times 15$, $96@10 \times 10$ and $2@1 \times 1$. The last CONV layer with $2@1 \times 1$ plays the same role as FC layers in our work.

The third project was also led by Dr. Cha [16] with region based DL method. This method achieved a 94.7%, 91.8%, 86.1%, 90.9% and 85.2% test accuracy respectively for concrete cracks, medium steel corrosion, high steel corrosion, bolt corrosion and steel delimitation. This network contains 16 layers, 5 of which are CONV layers with the size of $96@7 \times 7$, $256@5 \times 5$, $384@3 \times 3$, $384@3 \times 3$ and $256@3 \times 3$ separately. Although it has a huge amount of parameters, the prediction of each image takes only 0.03 seconds due to the matching performance of the computer. Therefore, this model can also process videos in real time.

Compared with all projects above, the results of our work consistent with the research showing that CNN method can detect concrete cracks with pretty high accuracy. Besides, our results suggest that this method also has a promising potential in detecting nature stone problem like cracks and joints detection. Last but not least, we have verified that transfer learning method makes training process performance better.

7.3 Limitations and Potential Improvements

The limitations of this work mainly refer to two aspects: the amount of images and the computing power of the machine. Our training set is rather small, which reduces the performance of models because there are only limited features extracted during training process. If we had a bigger data base, we would have higher test accuracy and a chance to train a model that can detect Multiple Damage Types. The other limitation, lack of machine computer power, affects the depth and complexity of the network structure and the application scene of trained model. With higher computer power, we can build a deeper network with better performance.

Chapter 8

Conclusions

The chapter summarizes contributions and makes conclusions of this work. Suggestions and preview for further researches are also presented.

8.1 Conclusion

The primary purpose of this thesis is to detect cracks of nature stones with CNN method. In order to achieve it, we build a dataset about concrete crack with 40,000 images in two classes. After analysis of classic CNN structures, a network structure is built and trained to detect concrete cracks. With the method of transfer learning, the modle that is able to detect cracks of nature stones is also trained. Moreover, with limited size of dataset, we build a model that can distinguish cracks from joints of nature stones.

To conclude, the proposed CNN method can not only detect crack but also distinguish joint from crack. In addition, the application of transfer learning can improve training efficiency and achieve higher accuracy of prediction with fewer training steps. This means that neural networks have greater potential in the field of architectural renovation. CNN has the ability to learn features from a huge amount of data, but it also means that the implementation of method requires a vast amount of training to build a robust model.

8.2 Outlook

If a database of nature stone with multiple defect types can be created, DL method would be able to identify more types of damage. Imaging there is a real time classifier that is able to mark all different types of stone damages from video inputs (see figure 8.1), this network for automatical identification of multiple damage types can mark the location and types of defects. It will provide us a work method which is safer, more efficient, with lower cost and labor costs, as well as with a more objective analysis in building renovation.

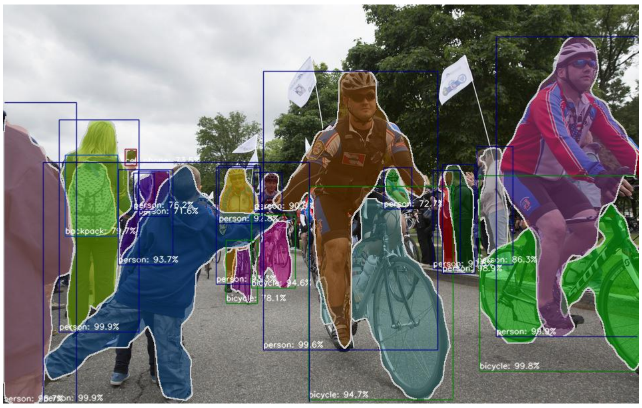


Figure 8.1: Mask RCNN

Bibliography

[1] CS231n convolutional neural networks for visual recognition.

[2] CSC321 winter 2018.

[3] Custom training: walkthrough | TensorFlow core.

[4] Data input pipeline performance | TensorFlow core.

[5] Digital noise: What is it & how to correct it.

[6] Illustrated glossary on stone deterioration =: Glossaire illustr sur les formes d'altration de la pierre. OCLC: 845610298.

[7] ImageNet.

[8] Intel core i7-8550u processor (8m cache, up to 4.00 GHz) product specifications.

[9] Kernel (image processing). Page Version ID: 873534713.

[10] Setting the learning rate of your neural network.

[11] tf.initializers.truncated_normal | TensorFlow core 1.13.

[12] tf.nn.convolution | TensorFlow core 1.13.

[13] What is deep learning? | how it works, techniques & applications.

[14] What is image recognition? - definition from WhatIs.com.

[15] Young-Jin Cha, Wooram Choi, and Oral Bykztrk. Deep learning-based crack damage detection using convolutional neural networks: Deep learning-based crack damage detection using CNNs. 32(5):361–378.

[16] Young-Jin Cha, Wooram Choi, Gahyun Suh, Sadegh Mahmoudkhani, and Oral Bykztrk. Autonomous structural visual inspection using region-based deep learning for detecting multiple damage types: Autonomous SHM using deep faster r-CNN. 33(9):731–747.

[17] Adit Deshpande. A beginner's guide to understanding convolutional neural networks.

[18] Xavier Glorot and Yoshua Bengio. Understanding the difculty of training deep feedforward neural networks. page 8.

[19] Google for Startups Campus Korea. Google tech talk with jeff dean at campus seoul.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification.

[21] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[23] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Ha. Gradient-based learning applied to document recognition. page 46.

[24] Fei-Fei Li, Justin Johnson, and Serena Yeung. Lecture 6: Training neural networks, part i. page 90.

[25] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks.

[26] Dmytro Mishkin and Jiri Matas. ALL YOU NEED IS a GOOD INIT. page 13.

[27] Natlia Neto and J. de Brito. Inspection and defect diagnosis system for natural stone cladding. 23(10):1433–1443.

[28] Satyen Rajpal. Crack detection for an autonomous UAV. contribute to satyenrajpal/concrete-crack-detection development by creating an account on GitHub. original-date: 2018-01-03T01:33:05Z.

[29] Sumit Saha. A comprehensive guide to convolutional neural networks the ELI5 way.

[30] Tarang Shah. About train, validation and test sets in machine learning.

[31] A. Silva, J. de Brito, and P.L. Gaspar. Service life prediction model applied to natural stone wall claddings (directly adhered to the substrate). 25(9):3674–3684.

[32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition.

[33] Dave Smith. Cutting-edge face recognition is complicated. these spreadsheets make it easier.

[34] Leslie N. Smith. Cyclical learning rates for training neural networks.

[35] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions.

[36] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision.

[37] Rohan Varma. Picking loss functions - a comparison between MSE, cross entropy, and hinge loss.

[38] Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, and Antonio Torralba. Object detectors emerge in deep scene CNNs.

Appendix A

Rename File with Batch

```
@echo off
title Batch Rename
echo.
echo With this batch file we could rename all files in this order.
echo.
echo.&set /p strtemp3= file typ:
echo.&set /p strtemp2= What you want to add befor the name:
setlocal enabledelayedexpansion
for /f "delims=" %%a in ('dir /a /b *.%strtemp3%') do (
ren "%%~a" "%strtemp2%_%%a")
echo.
echo OK
echo.
pause
```

Appendix B

Input

```
# coding: utf-8

# In[1]:

import tensorflow as tf
import os
import numpy as np

def get_files(file_dir):
    Positives = []
    label_Positives = []
    Negatives = []
    label_Negatives = []
    for file in os.listdir(file_dir):
        name = file.split(sep='.')
        if 'Positive' in name[0]:
            Positives.append(file_dir + file)
            label_Positives.append(0)
        else:
            if 'Negative' in name[0]:
                Negatives.append(file_dir + file)
                label_Negatives.append(1)
        image_list = np.hstack((Positives,Negatives))
```

```
        label_list = np.hstack((label_Positives,label_Negatives))
            # print('There are %d Positives\nThere are %d Negatives' %(len(
    Positives), len(Negatives)))
    temp = temp.transpose()
    np.random.shuffle(temp)
    image_list = list(temp[:,0])
    label_list = list(temp[:,1])
    label_list = [int(i) for i in label_list]
    return image_list,label_list

def get_batch(image,label,image_W,image_H,batch_size,capacity):
    image = tf.cast(image,tf.string)
    label = tf.cast(label, tf.int32)
    input_queue = tf.train.slice_input_producer([image,label])
    label = input_queue[1]
    image_contents = tf.read_file(input_queue[0])
    image = tf.image.decode_jpeg(image_contents,channels =3)
    image = tf.image.resize_image_with_crop_or_pad(image, image_W, image_H)
    image = tf.image.per_image_standardization(image)
    image_batch, label_batch = tf.train.batch([image, label],batch_size =
    batch_size, num_threads = 8, capacity = capacity)
    label_batch = tf.reshape(label_batch , [batch_size])
    image_batch = tf.cast(image_batch,tf.float32)
    return  image_batch, label_batch
```

Appendix C

Network Structure

```
#coding=utf-8
import tensorflow as tf

def inference(images, batch_size, n_classes):

    with tf.variable_scope('conv1') as scope:
        weights = tf.get_variable('weights',
                                  shape=[3, 3, 3, 16],
                                  dtype=tf.float32,
                                  initializer=tf.truncated_normal_initializer(stddev=0.1, dtype=tf.float32))
        biases = tf.get_variable('biases',
                                 shape=[16],
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0.1))
        conv = tf.nn.conv2d(images, weights, strides=[1, 1, 1, 1], padding='SAME')
        pre_activation = tf.nn.bias_add(conv, biases)
        conv1 = tf.nn.relu(pre_activation, name=scope.name)

    with tf.variable_scope('pooling1_lrn') as scope:
            pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1], padding='SAME', name='pooling1')
```

```
            norm1 = tf.nn.lrn(pool1, depth_radius=4, bias=1.0, alpha=0.001 / 9.0, beta=0.75, name='norm1')

    with tf.variable_scope('conv2') as scope:
                weights = tf.get_variable('weights',
                                          shape=[3, 3, 16, 32],
                                          dtype=tf.float32,
                                          initializer=tf.truncated_normal_initializer(stddev=0.1, dtype=tf.float32))
                biases = tf.get_variable('biases',
                                         shape=[32],
                                         dtype=tf.float32,
                                         initializer=tf.constant_initializer(0.1))
                conv = tf.nn.conv2d(norm1, weights, strides=[1, 1, 1, 1], padding='SAME')
                pre_activation = tf.nn.bias_add(conv, biases)
                conv2 = tf.nn.relu(pre_activation, name='conv2')

    # pool2 and norm2
    with tf.variable_scope('pooling2_lrn') as scope:
        norm2 = tf.nn.lrn(conv2, depth_radius=4, bias=1.0, alpha=0.001 / 9.0, beta=0.75, name='norm2')
        pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1], padding='SAME', name='pooling2')

    with tf.variable_scope('local3') as scope:
        reshape = tf.reshape(pool2, shape=[batch_size, -1])
        dim = reshape.get_shape()[1].value
        weights = tf.get_variable('weights',
                                  shape=[dim, 128],
                                  dtype=tf.float32,
                                  initializer=tf.truncated_normal_initializer(stddev=0.005, dtype=tf.float32))
        biases = tf.get_variable('biases',
```

```
                                  shape=[128],
                                  dtype=tf.float32,
                                  initializer=tf.constant_initializer(0.1))
    local3 = tf.nn.relu(tf.matmul(reshape, weights) + biases, name=scope.name)

    # local4
    with tf.variable_scope('local4') as scope:
        weights = tf.get_variable('weights',
                                  shape=[128, 128],
                                  dtype=tf.float32,
                                  initializer=tf.truncated_normal_initializer(stddev=0.005, dtype=tf.float32))
        biases = tf.get_variable('biases',
                                 shape=[128],
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0.1))
        local4 = tf.nn.relu(tf.matmul(local3, weights) + biases, name='local4')

    # softmax
    with tf.variable_scope('softmax_linear') as scope:
        weights = tf.get_variable('softmax_linear',
                                  shape=[128, n_classes],
                                  dtype=tf.float32,
                                  initializer=tf.truncated_normal_initializer(stddev=0.005, dtype=tf.float32))
        biases = tf.get_variable('biases',
                                 shape=[n_classes],
                                 dtype=tf.float32,
                                 initializer=tf.constant_initializer(0.1))
        softmax_linear = tf.add(tf.matmul(local4, weights), biases, name='softmax_linear')

    return softmax_linear

def losses(logits, labels):
```

```
    with tf.variable_scope('loss') as scope:
        cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits \
                        (logits=logits, labels=labels, name='
   xentropy_per_example')
        loss = tf.reduce_mean(cross_entropy, name='loss')
        tf.summary.scalar(scope.name + '/loss', loss)
    return loss

def trainning(loss, learning_rate):
    with tf.name_scope('optimizer'):
        optimizer = tf.train.AdamOptimizer(learning_rate= learning_rate)
        global_step = tf.Variable(0, name='global_step', trainable=False)
        train_op = optimizer.minimize(loss, global_step= global_step)
    return train_op

def evaluation(logits, labels):
    with tf.variable_scope('accuracy') as scope:
        correct = tf.nn.in_top_k(logits, labels, 1)
        correct = tf.cast(correct, tf.float16)
        accuracy = tf.reduce_mean(correct)
        tf.summary.scalar(scope.name + '/accuracy', accuracy)
    return accuracy
```

Appendix D

Train the model

```
# coding: utf-8

# In[1]:
import os
import numpy as np
import tensorflow as tf
import input_data
import model


N_CLASSES = 2
IMG_W = 220
IMG_H = 220
BATCH_SIZE = 32
CAPACITY = 256
MAX_STEP = 2000
learning_rate = 0.0001


def run_training():

    train_dir = 'C:\\Users\\Jiesheng Yang\\Desktop\\8thTry\\Train\\'
    logs_train_dir = 'C:\\Users\\Jiesheng Yang\\Desktop\\8thTry\\
```

```
    LeNet5_model_of_crack\\'
    train, train_label = input_data.get_files(train_dir)
    train_batch, train_label_batch = input_data.get_batch(train,
                                                          train_label,
                                                          IMG_W,
                                                          IMG_H,
                                                          BATCH_SIZE,
                                                          CAPACITY)
    train_logits = model.inference(train_batch, BATCH_SIZE, N_CLASSES)
    train_loss = model.losses(train_logits, train_label_batch)
    train_op = model.trainning(train_loss, learning_rate)
    train__acc = model.evaluation(train_logits, train_label_batch)
    summary_op = tf.summary.merge_all()
    sess = tf.Session()
    train_writer = tf.summary.FileWriter(logs_train_dir, sess.graph)
    saver = tf.train.Saver()
    sess.run(tf.global_variables_initializer())
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)

    ckpt = tf.train.get_checkpoint_state(logs_train_dir)
    if ckpt and ckpt.model_checkpoint_path:
    saver.restore (sess,ckpt.model_checkpoint_path)

    try:
        for step in np.arange(MAX_STEP):
            if coord.should_stop():
                    break
            _, tra_loss, tra_acc = sess.run([train_op, train_loss, train__acc])

            if step % 50 == 0:
                print('Step %d, train loss = %.2f, train accuracy = %.2f%%' %(step, tra_loss, tra_acc*100.0))
                summary_str = sess.run(summary_op)
                train_writer.add_summary(summary_str, step)
```

```

            if step % 100 == 0 or (step + 1) == MAX_STEP:
                checkpoint_path = os.path.join(logs_train_dir, 'model.ckpt')
                saver.save(sess, checkpoint_path, global_step=step)

    except tf.errors.OutOfRangeError:
        print('Done training -- epoch limit reached')
    finally:
        coord.request_stop()
    coord.join(threads)
    sess.close()

# train
run_training()
```

Appendix E

Usage

```
# coding: utf-8

# In[1]:
#coding=utf-8
import tensorflow as tf
from PIL import Image
import matplotlib.pyplot as plt
import input_data
import numpy as np
import model
import os

def get_one_image(train):
    files = os.listdir(train)
    n = len(files)
    ind = np.random.randint(0,n)
    img_dir = os.path.join(train, files[ind])
    image = Image.open(img_dir)
    plt.imshow(image)
    plt.show()
    image = image.resize([220, 220])
    image = np.array(image)
    return image
```

```


def evaluate_one_image():

#    train = 'C:\\Users\\Jiesheng Yang\\Desktop\\Wall\\'
    train = 'C:\\Users\\Jiesheng Yang\\Desktop\\8thTry\\Test\\'
    image_array = get_one_image(train)

    with tf.Graph().as_default():
        BATCH_SIZE = 1
        N_CLASSES = 2
        image = tf.cast(image_array, tf.float32)
        image = tf.image.per_image_standardization(image)
        image = tf.reshape(image, [1, 220, 220, 3])
        logit = model.inference(image, BATCH_SIZE, N_CLASSES)
        logit = tf.nn.softmax(logit)
        x = tf.placeholder(tf.float32, shape=[220, 220, 3])
        logs_train_dir = 'C:\\Users\\Jiesheng Yang\\Desktop\\8thTry\\LeNet5_model_of_crack\\'
        saver = tf.train.Saver()

        with tf.Session() as sess:
            print("Loading Model from the specified directory")
            ckpt = tf.train.get_checkpoint_state(logs_train_dir)
            if ckpt and ckpt.model_checkpoint_path:
                global_step = ckpt.model_checkpoint_path.split('/')[-1].split('-')[-1]
                saver.restore(sess, ckpt.model_checkpoint_path)
                print('Model Loading successful, Training step is %s' % global_step)
            else:
                print('Load Model failed, file not found')
            prediction = sess.run(logit, feed_dict={x: image_array})
            max_index = np.argmax(prediction)
            if max_index==0:
```

```
                print('The Probability of a Crack is %.6f' %prediction[:, 0])
            else:
                print('The Probability of No Crack is %.6f' %prediction[:, 1])

evaluate_one_image()
```