Deep Learning Based Multi-Sensor Integration for Pavement Crack Detection

by Dr. Mingxuan Sun and Dr. Xiangwei Zhou

School of Electrical Engineering and Computer Science

Louisiana State University and

Agricultural and Mechanical College

Final Report

LTRC Project No. 20-1TIRE

Conducted for

Louisiana Transportation Research Center

July 2020

Abstract

Crack detection using surficial images has been an important task in structural engineering. Recently, deep neural networks are being utilized extensively for image classification and pattern recognition. Deep neural networks are powerful tools that automatically extract different levels of image features and generate accurate predictions. This study investigates crack image classification using different deep neural network models such as variations of VGG16 along with traditional support vector machines (SVM). We compare the results of these models from both prediction accuracy and computational efficiency. The dataset utilized for this study consists of 2,255 concrete surface images with and without cracks. In our study, we find that particular deep neural network models perform better than traditional SVM in crack detection. The best experiment using a variation of VGG-16 achieves higher accuracy, which demonstrates the potential of deep learning models.

Acknowledgments

We would like to thank Mr. Abhishek Huddar for his great contribution to this project.

Contents

List of Figures

1 Introduction

In the field of structural engineering, for monitoring, inspection and maintenance there is a great demand for bringing in automation. In the United States, the federal law enforces routine inspections every two years for both buildings as well as bridges. So it is imperative to make sure that these quality standards are met. This is exactly where image processing comes into the picture. Using deep convolutional neural networks, the images of bridges and buildings taken from unmanned aerial devices can be processed at a much faster rate in identification and detection of whether cracks are visible over the surfaces, which will, in turn, help the structural engineers to make necessary fixes to the structure by making sure the strength of the structure is maintained.

The deep learning convolutional neural networks are a type of artificial neural network which has an input and an output layer, and in between they have various filtering layers, where each layer absorbs specific detail of an image, like angle, orientation, color, etc. Millions of images are sent through these layers, which will train the image to identify better. There are various ways in which the neural network will identify whether there is the presence of cracks in any given image or not: one is by labeling the images cracked and non-cracked, the second method is to do classification, and the third method would be a process called localization (where the exact area of the crack is located). In this project, we use the method of classification to classify whether the images have cracks or not. Here the efficiency of our network identifying the images depends on various parameters, such as the way the layers have been arranged along with how big the dataset to be trained is.

This particular section showcases how our models operate at a high level to generate the required accuracy. As shown in Fig. 3, we see both the training procedure and the testing procedure. The training procedure has been represented by solid demarcations, and the testing procedure has been represented by dotted demarcations. The data that is being used to input in the CNN classifiers are a bunch of surface images of concrete buildings, bridges, and decks which have been taken by a DSLR camera. The data include images of all kinds; for example, there are images which have shadows in it, and there are some images which have very poor visibility of concrete cracks which might ultimately provide false positive values. These kinds of images have been deliberately included in the training and the testing set. In our dataset, any image which can be differentiated from a normal human eye as cracked/non-cracked has only been utilized. In totality, our dataset has 2,255 raw images of resolution 128 X 128.

In the images below, we can see how the concrete crack images have been segregated and processed. Most of these images have been cropped out to 128 X 128 pixels resolutions to keep the integrity of the dataset. For a few of the images, only the part of an image that contains the crack has been carefully chosen. The dataset images below show exactly the kind of images that have been chosen for this experiment.

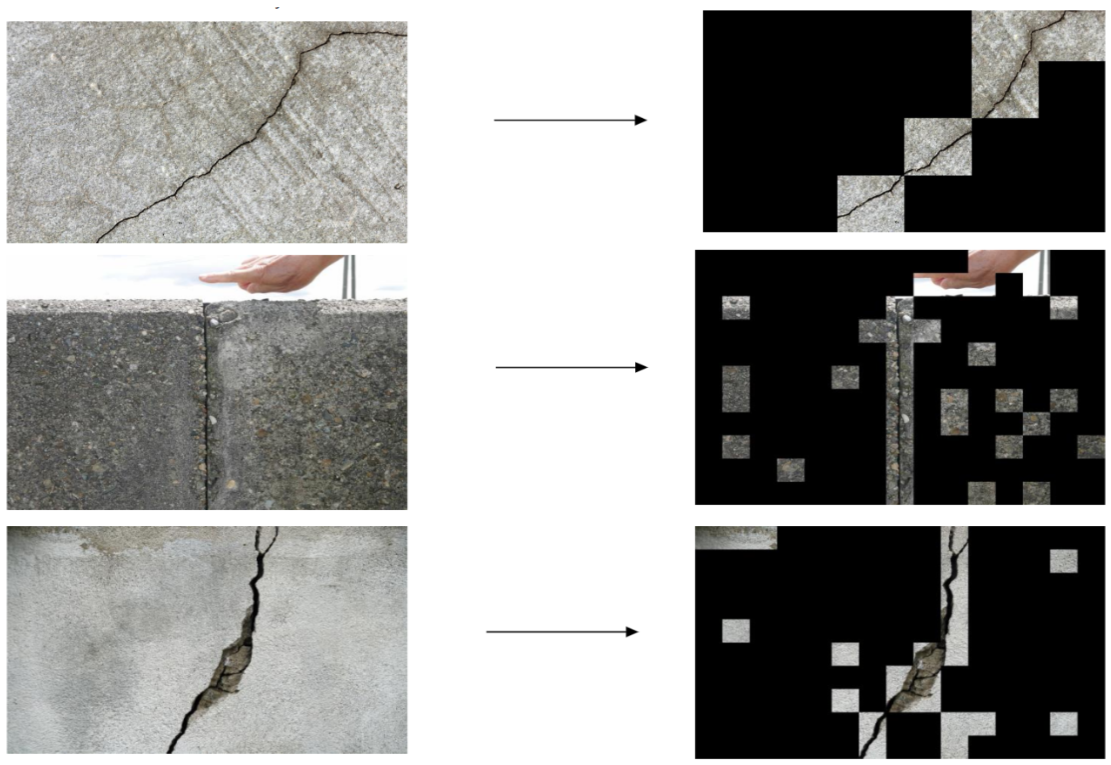


Figure 1: Raw images to the left, output classification to the right.

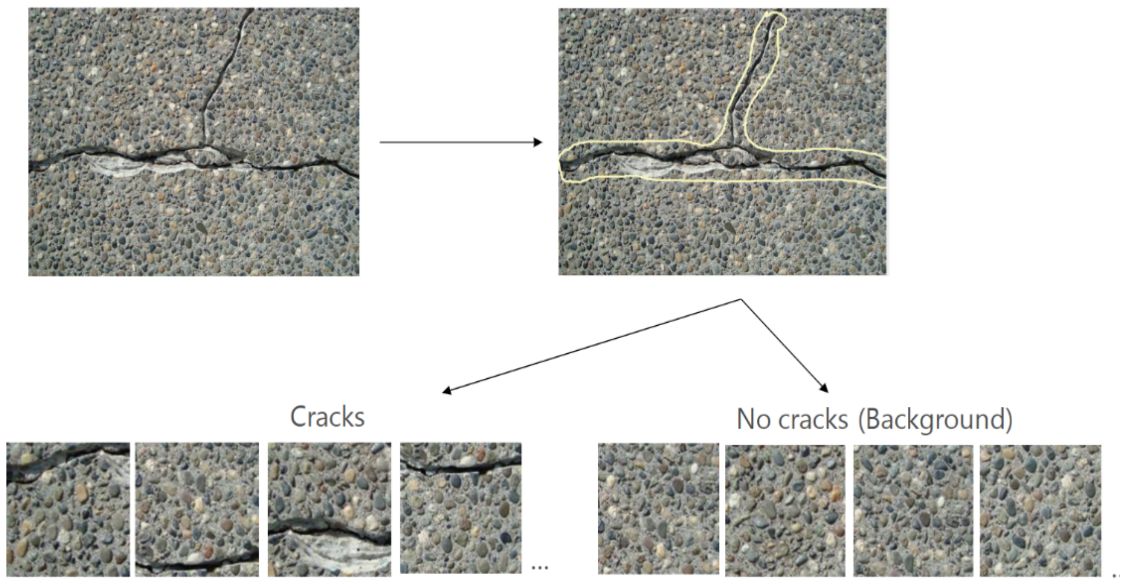


Figure 2: Some example images used for training.

These 2,255 images are split out and are strategically placed in two folders namely cracked and no- cracked. The cracked folder contains the data (images) that are required for training, and similarly, the non-cracked folder has all the data (images) required for training. However, in our dataset design, the folders cracked and non-cracked also contain sub-folders that have images for testing and validation. The cracked folder, which is our 1st classifier contains 958 training images and 33 testing images. Similarly, the non-cracked folder contains 1,189 training images and 75 testing images. Now the dataset that has been prepared is given as input to the CNN classifier to segregate the cracked images from the no-cracked ones. To summarize this proposed methodology, we have raw images as training data from which a dataset has been prepared. This is further split into training and testing/validation data. This split is trained on the CNN classifier and post-training, raw images are given as an input to the CNN classifier and the accuracy of the CNN is calculated.

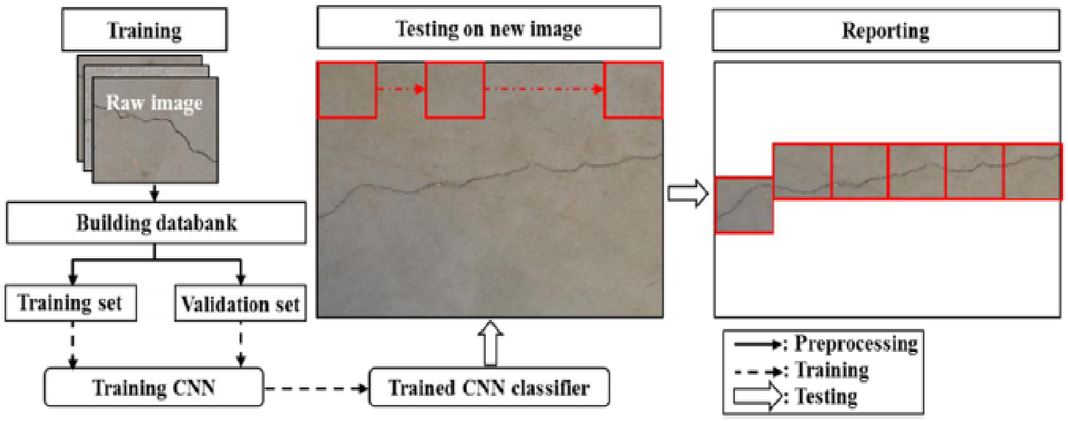


Figure 3: Concrete crack detection flowchart.

2 Related Work

Before experiments were carried out, various work in the related field of structural engineering has been studied and collated. These studies have utilized the applications of different deep learning models and they stand as the foundation for all the 3 experiments that have been carried out. Some of these models have used support vector machines, whereas most of these have some form of deep learning neural network tested. Even though some of the experiments seem to yield far lesser accuracy than the accuracy achieved in our model, it needs to be noted that most of these studies are applications in nature and have done comparative studies to pick the best neural network for the respective datasets that were utilized. Almost all of the studies have used concrete crack images taken from buildings and bridges. A few of the studies have used road pavement cracks for classification, however, from an implementation point of view, there is a necessity to go through these studies as the models used in our experiments are intended to produce higher accuracy and generate as few false positives as possible.

A 2016 study utilized deep convolutional neural networks to classify whether cracks are present or not on roads (Zhang, Yang, Zhang, & Zhu, 2016). A relatively smaller dataset was used to perform this study, where the dataset included around 500 images of roads. Here these images were of size 3264 X 2448 and all these images were taken from a smartphone. The objective of the study was to show whether the given image of the pavement did indeed have any cracks or not. ConvNet was the deep learning model utilized for the entirety of this study. The neural network consisted of considerably fewer layers making it less deep. The architecture had 4 convolutional layers followed by 2 fully connected layers. ReLU is the activation function used for this study just like most others. Also, stochastic gradient descent (SGD) was used for training where individual batch was consisted of 48 images and used 20 epochs. Not just a neural network study, but also an SVM was used for comparison and to show the superiority of using deep learning. It was seen post experimentation that the deep learning ConvNet model reached a precision of 0.86 and SVM showed a lower side of precision with 0.81. The recall was drastically higher for ConvNet at 0.93; however, SVM failed to provide a higher recall and reached the maximum value of 0.67. Overall the ConvNet achieved an F1 score of 0.89 coming out as the better model for accurate classification.

A conference paper published in the year 2018 did a study on the feasibility of using deep learning convolutional neural networks for structural inspection (Dorafshan, Thomas, Coopmans, & Maguire, 2018). Two modes were used for assessing whether a deep learning model can be incorporated for this or not. These two modes are fully trained and transfer learning. This paper mentioned AlexNet as its primary model against which various other models are tested. This architecture is more detailed in structure and much deeper than the ConvNet architecture. It consists of in total 5 convolutional layers, followed by 3 fully connected layers. ReLU is the activation layer used, along with a softmax and a classification layer. The study was done on 3 sets of data, which contained images taken from 3 cameras with varying resolutions. Data collected from the Nikon camera (16 MP) were of resolution 4068 X 3456. Secondly, they used a GoPro (12 MP) with the resolution of images being 4000 X 3000 and finally DJI Mavic camera of the same resolution and dimensions as that of GoPro. Overall it was seen that the dataset which consisted of higher resolution images yielded better testing accuracy, where the FT and TL stood at 79 percent and 89 percent. It was concluded that in using deep learning models, clarity and resolution of datasets made more impact rather than any particular model.

A performance-based study involves various deep learning convolutional networks for detecting cracks in concrete bridges and building (Özgenel & Sorguç, 2018). This study did not introduce any novel deep learning model, rather ran their dataset on 7 pre-trained neural networks namely AlexNet, ResNet, VGG-16, VGG-19, GoogleNet, ResNet 50, ResNet 101, and ResNet 152. It was observed that for smaller data size, VGG-16 and GoogleNet yielded 96 percent accuracy; however for a bigger dataset of over 14,000 images ResNet 101 and ResNet 152 showed an overwhelming accuracy of 97 percent.

Apart from looking at studies involving only neural networks, it is important to look at other aspects of the network build such as edge detectors. A 2019 study was performed solely on using various edge detectors both spatial and frequency based (Dorafshan, Thomas, & Maguire, 2019). The experiment used 4 spatial edge detectors namely Roberts, Prewitt, Sobel, and Laplacian of Gaussian(LOC). Along with these 2, frequency-based edge detectors were also studied which are Butterworth and Gaussian. By far LOG resulted in an accuracy of 92 percent which is much higher than any other edge detector. Furthermore, an interesting study was also taken into consideration before moving ahead with the experiments. This study was done in the year 2012, which considered a deep learning neural network with varying layers to assess the error rate (Krizhevsky, Sutskever, & Hinton, 2012). The study considered images from ImageNet repository, in which around 15000 images were used for training and testing. It was seen that out of the 3 models trained and tested (SIFT, Sparse Coding, and CNN), CNN achieved the least error rate with tier-1 generating a 37.5 percent error rate and tier-5 generating a much lesser error rate of 17 percent.

3 Support Vector Machines

Support vector machines are one of the ways in machine learning where images can be classified and tested along with training accuracy plotted out (Moussa & Hussain, 2011). A support vector machine is also a kind of supervised learning model. In the data preprocessing stage, whichever features need to be used are segregated and are usually mapped on to a higher dimensional space to successfully distinguish the images that have a crack from the non-cracked ones. What SVM does is, it identifies a bunch of points in every class which is closer to the other classes. Once it does so, the SVM calculates a hyper-plane that distinguishes between the classes. This hyper-plane that is plotted out is usually called the maximum-margin hyper-plane and it is what makes the SVM scalable and robust. After the process of training is performed by the SVM, the only part left would be the testing. In testing, all the SVM does is to try to accurately categorize which class the images belong to, either cracked or non-cracked. The mathematical implementation of support vector machines is given below.

Initially we will define a particular hyper-plane (Gareth, 2010)

$$x : f(x) = xw + w_0 = 0$$

The output that we derive out of this classifier is actually defined by the following only in case where x has already been given

$$G(x) = sign(f(x))$$

Now let us assume that we have a training dataset of size n, i.e., (x1,y1)....(Xn,Yn)

$$f(x_i) >= 1, \text{if } y_i = 1$$

$$f(x_i) >= -1, \text{if } y_i = -1$$

We shall denote the hyper-planes as follows $H_0 = x : f(x) = 1$ and $H_1 = x : f(x) = -1$. Here when we talk about the margin, it is the space between both of the hyper-planes H_0 and H_1. So what the SVM does is, it will identify the hyper-planes with respect to the hyper-plane with the biggest margin. The diagramatic representation of what a hyper-plane looks like is provided in the image below, where we can see how SVM uses the concept of a hyper-plane. In the figure, we can see two symbols, squares and circles. Here the squares represent class 1 and the circles represent class 2. Here both of these classes are classified by the hyper-plane function : wx-b=0.

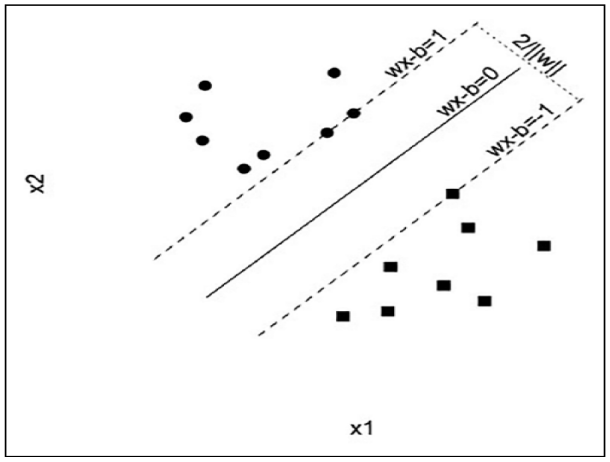


Figure 4: The hyper-plane margin. Two classes of points: circles and squares classified by hyper-plane wx-b=0.

When using SVM as a classifier, it is normal to encounter a linear hyperplane between the two classes. But this particular feature does not need to be manually implemented to have a hyperplane, instead a technique of SVM known as kernel trick can be utilized. So here we have an SVM kernel which essentially takes up a low dimensional input into a higher dimensional space. Then the inseparable problem is transformed into a separable problem. Coming to the implementation aspect of the support vector machine, Python provides an built-in library known as sci-kit learn. So in brief, the sci-kit library needs to be implemented first, followed by the creation of the object, performing all the necessary model fitting and in the end prediction is done. In essence, an SVM is an algorithm primarily utilized for the process of classification.

4 Deep Learning Models

Two different deep learning models have been used here to show which could be a better fit in this scenario. Both the deep learning models have the same type of layers and the exact activation functions. In this section, we will see all the layers that have been used to build this CNN architecture. The overall layers consist of an input layer, multiple convolutional layers, max-pooling layers, activation layer, fully connected layers, batch normalization and finally an output layer. We call any neural network as a deep convolutional neural network only when the particular network is composed of multiple layers. Before proceeding with the explanation for the different deep learning models used in the experiments, let us see the kind of images used for our dataset. Below are the examples of cracked and non-cracked data used in the experiments.

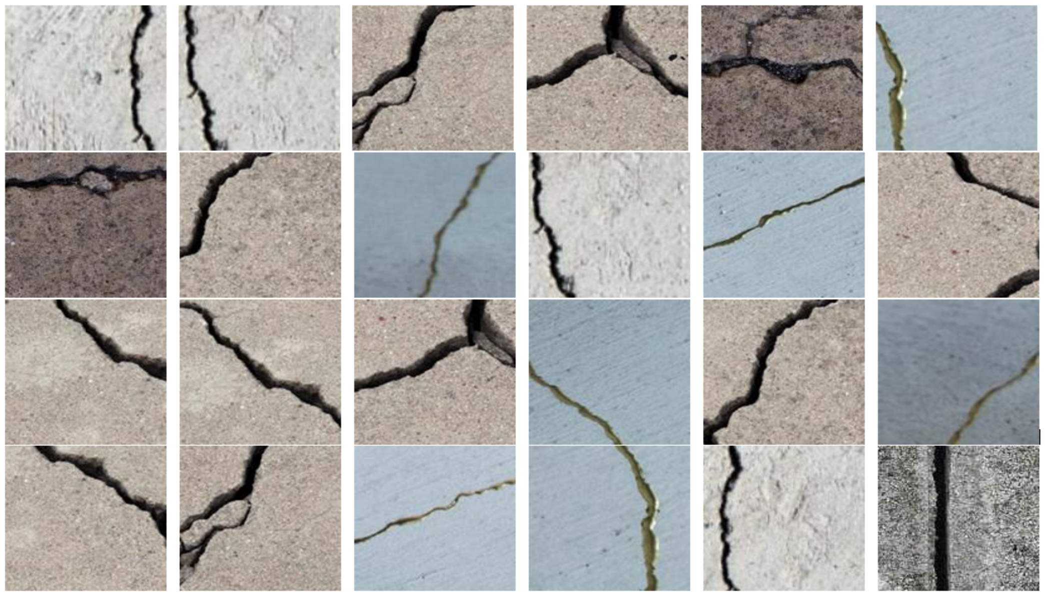


Figure 5: Cracked images from the dataset.

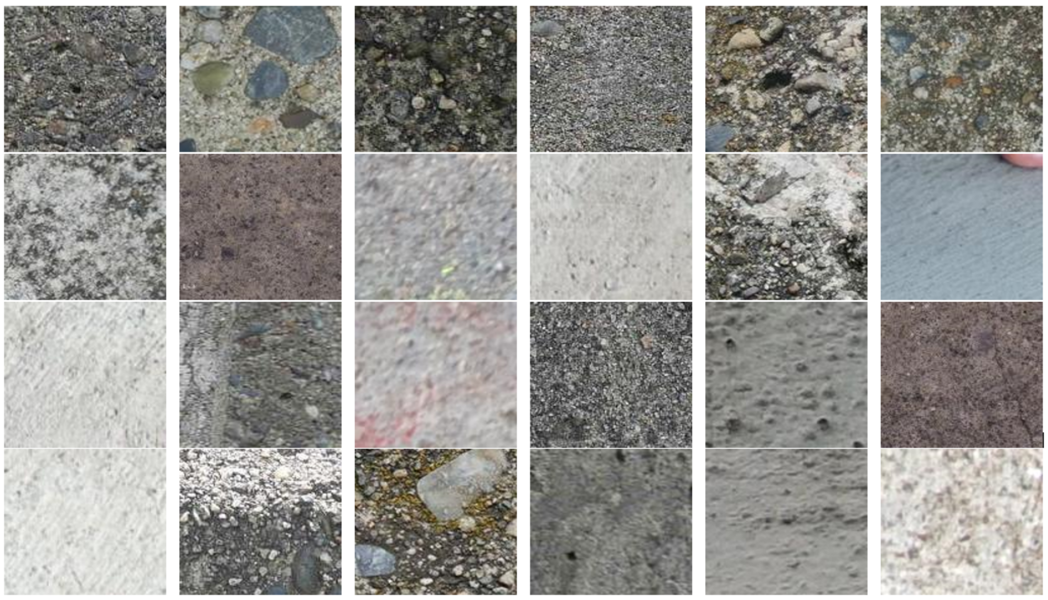


Figure 6: Non-cracked images from the dataset.

4.1 Overall Architecture of Both Models

In our architecture, we have an input layer which takes in the images of resolution 128 X 128 as input. Here the input images are all RGB images. These images are further generalized to a resolution of 1 X 1 X 96 in the L5 layer. Now, these 96 images as vectors are loaded onto the activation layer (Rectified Linear Unit), about which explanation is provided in the corresponding section. Post which we have a softmax layer, which is predominantly used to predict if there is crack or not (Cha, Choi, & Büyüköztürk, 2017). This happens after the 4th convolutional layer. Our architecture also holds batch normalization in layers 1, 3, and 5. All the layers' functionalities are explained in the following subsections. Below is the breakdown of all the layers and their respective features in Table 1.

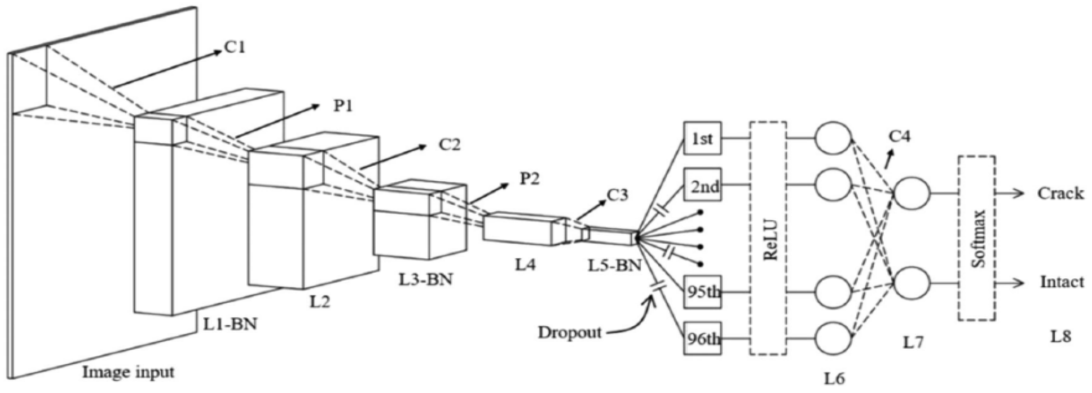


Figure 7: Overall architecture of the MatConvNet.

Table 1: Dimensions of the layers.

Layer	Height	Width	Depth	Operator	Height	Width	Depth	No	Stride
Input	128	128	3	C1	20	20	3	24	2
Layer 1	64	64	24	P1	7	7	-	-	2
Layer 2	32	32	24	C2	15	15	24	48-	2
Layer 3	16	16	48	P2	4	4	-	-	2
Layer 4	8	8	48	C3	10	10	48	96	2
Layer 5	1	1	96	ReLU	-	-	-	-	-
Layer 6	1	1	96	C4	1	1	96	2	1
Layer 7	1	1	2	Softmax	-	-	-	-	-
Layer 8	1	1	2	-	-	-	-	-	-

In comparison with the above model, a VGG-16 deep learning model is built with similar activation functions and layers. However, this VGG-16 has a different number of convolutional, pooling and fully connected layers. Below is a diagrammatic representation of VGG-16 architecture.

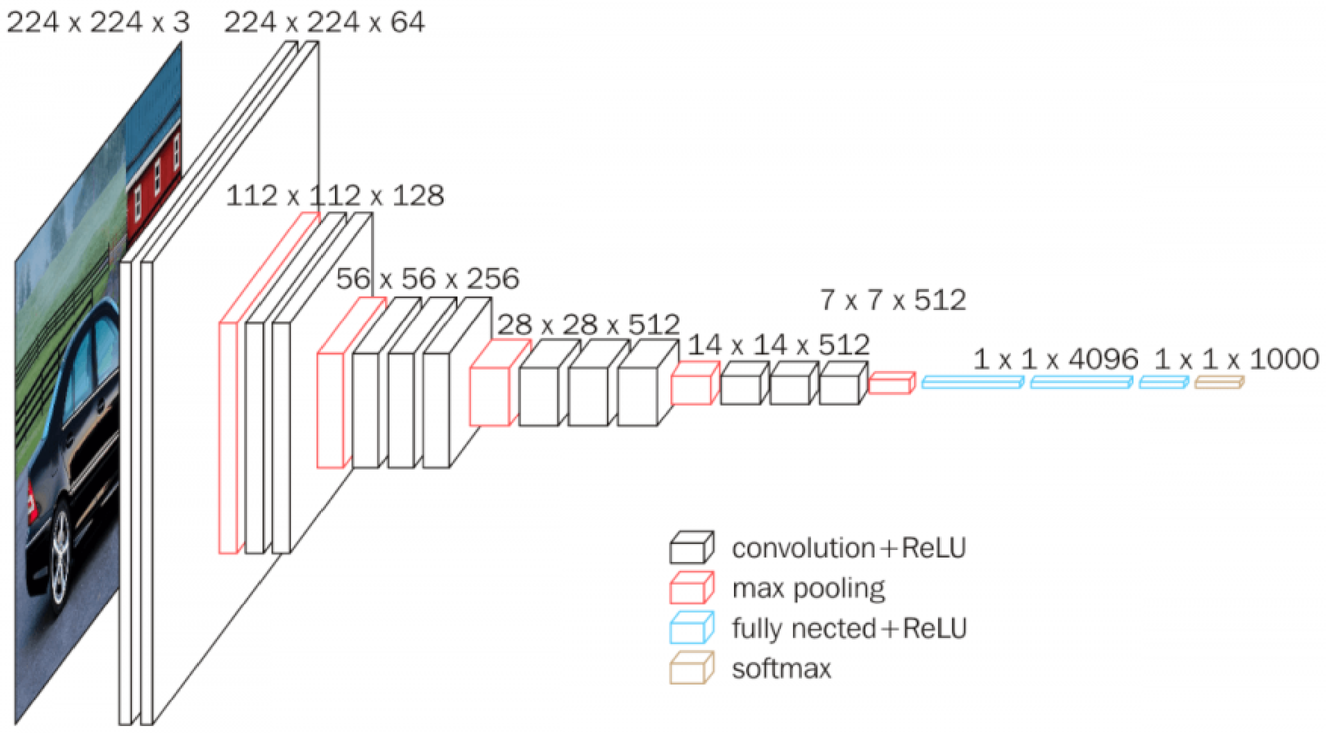


Figure 8: VGG-16 network architecture.

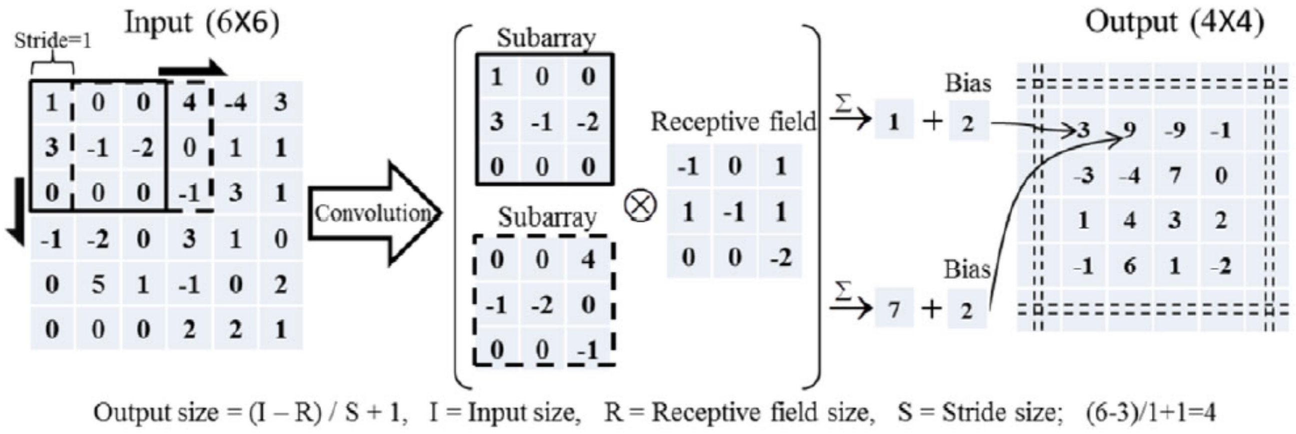
As we can see from the architecture model above, there are various convolutional layers. For the first two convolutional layers, we have an associated max pool layer. In the table below, we can see the complete setup of how the layers have been arranged, along with the filter and stride sizes. The output shape feature is how the dimensions of the image are modified as it approaches and passes through various layers and filters. It is a typical feature of neural networks where the size of the image is resized in every layer to extract the most of the features, and alongside the number of filters keeps on increasing. At the end of the network, we have 3 dense layers. All the 3 dense layers use ReLU activation. Creating the softmax layer would be the final one in the neural network before it could be trained. VGG-16 is one of the most complex and huge neural networks with over 138 million parameters.

Table 2: VGG layers dimensions.

Layer (type)	**Output Shape**	**Param Number**
conv2d_1 (Conv2D)	(None, 128, 128, 64)	1792
conv2d_2 (Conv2D)	(None, 128, 128, 64)	36928
max_pooling2d_1 (MaxPooling2	(None, 64, 64, 64)	0
conv2d_3 (Conv2D)	(None, 64, 64, 128)	73856
conv2d_4 (Conv2D)	(None, 64, 64, 128)	147584
max_pooling2d_2(MaxPooling2	(None, 32, 32, 256)	0
conv2d_5 (Conv2D)	(None, 32, 32, 256)	295168
conv2d_6(Conv2D)	(None, 32, 32, 256)	590080
conv2d_7 (Conv2D)	(None, 32, 32, 256)	590080
max_pooling2d_3 (MaxPooling2	(None, 16, 16, 256)	0
conv2d_8 (Conv2D)	(None, 16, 16, 512)	1180160
conv2d_9 (Conv2D)	(None, 16, 16, 512)	2359808
conv2d_10 (Conv2D)	(None, 16, 16, 512)	2359808
max_pooling2d_4 (MaxPooling2	(None, 8, 8, 512)	0
conv2d_11(Conv2D)	(None, 8, 8, 512)	2359808
conv2d_12 (Conv2D)	(None, 8, 8, 512)	2359808
conv2d_13 (Conv2D)	(None, 8, 8, 512)	2359808
max_pooling2d_5 (MaxPooling2	(None, 4, 4, 512)	0
flatten_1 (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 4096)	33558528
dense_2 (Dense)	(None, 4096)	16781312
dense_3 (Dense)	(None, 2)	8194

4.2 Convolutional Layer

The convolutional layer is responsible for 3 main operations in the architecture: initially, it does a dot product (matrix multiplication) of the given input array and the filter. Here the weights associated with the filter are randomly populated. There is the usage of stochastic gradient descent (SDC) for fine-tuning of the training values. We should also make a note that the size of the filter is the same as that of the size of the sub-array and the size of the filter is smaller than that of the size of the original input array. After we have done with the dot product, the values obtained from the dot product are added along with the bias. The primary reason for using convolutional layers is to reduce the amount of computation cost and also to reduce the size of the input data. Below is the image showing how the dot product generated is summed. This is specifically what happens in the convolutional layer.

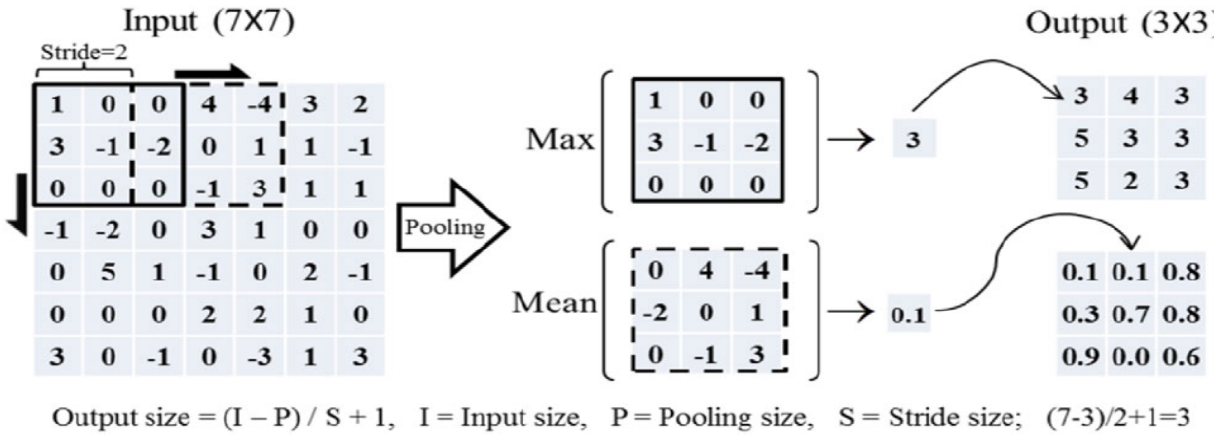


Input (6X6)
Stride=1
Convolution
Subarray
Receptive field
Subarray
Bias
Bias
Output (4X4)
Output size = (I – R) / S + 1, I = Input size, R = Receptive field size, S = Stride size; (6-3)/1+1=4

Figure 9: Convolution example.

4.3 Pooling Layer

One of the extremely important layers to include in any convolutional neural network is the pooling layer. The primary purpose of utilizing a pooling layer is to reduce the size of the given input array. This entire process of reducing the overall size of the input array is called downsampling. There are multiple ways of using the pooling layers. In our model, we have used the process of max-pooling, in which the layer takes up the maximum elements from the subarray of the input array. A study (Scherer, Müller, & Behnke, 2010) showed that utilizing the process of max-pooling is the best when using images as our datasets. As a reference from this study, for this project, all the pooling layers that have been chosen are max-pooling layers. Below is a diagrammatic representation of the max-pooling layer.



Input (7X7)
Stride=2
Pooling
Max
Mean
Output (3X3)
Output size = (I – P) / S + 1, I = Input size, P = Pooling size, S = Stride size; (7-3)/2+1=3

Figure 10: Pooling example.

4.4 ReLU Layer

ReLU is the activation layer that we will be using in our CNN model. One more 2010 study (Nair & Hinton, 2010) claimed that whenever we use sigmoid based functions, the saturating nonlinearities will slow down the computation process. Because of this, ReLU is being used as a non-linear function in this particular model. In the figure below, we could see various non-linear functions. Here ReLU has no bounds when it comes to the value for output; however, the input values here cannot be negative. As ReLU has zeroes and ones (0,1) as its gradients, it avoids a lot of complex sigmoid functions and hence makes the computation much easier and faster assisting in providing superior accuracy.

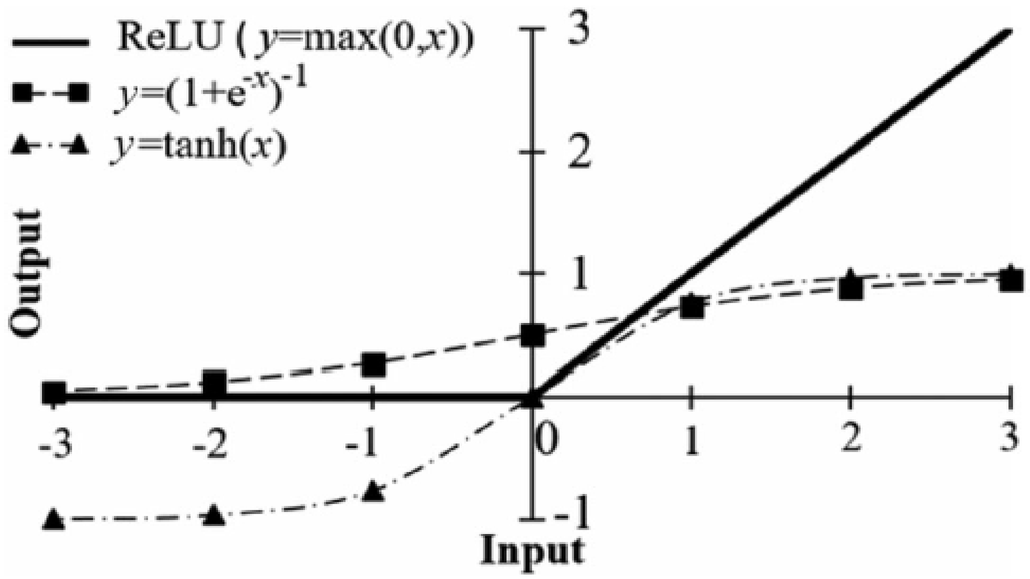


Figure 11: ReLU non-linear functions.

4.5 Softmax Layer

Softmax is one of the final layers in any given deep learning architecture. This is primarily utilized for processing the input data. This layer puts out probabilities of outcomes in the form of a vector. However, we need to keep in mind that the softmax layer might get costly when more classes are used. In such a situation, we could always use a concept known as candidate sampling which will not consider all the classes but will limit its computations only to a few specific classes. It also needs to be kept in mind that a softmax layer usually just considers 1 member per given class and in the situations where an object belonging to multiple numbers of classes, this will not work with the softmax layer. If we encounter such a situation, we could always use logistic regression.

5 Experiments and Results

After pre-processing the dataset and creating a databank that is suitable enough to run on various models, the process of training and testing has been performed. There are a total of 2,255 images. We random split the data such that 10 percent of the data is used for testing and the rest of the 90 percent of the data is used for training. We repeat the process five times and save the data partitions as five folds. Cross-validation is performed. Specifically, the model runs five times and each time it runs on a different data fold. The model accuracy is averaged over five folds. All three different models run on the same data partitions. The first model is a Support Vector Machine (SVM), the second one and the third one are variants of VGG-16 models. Here the variant means, different activation layers and the different number of convolutional layers. All the 3 models have been implemented using Python.

5.1 Image Classification using Support Vector Machines

The input data folder consists of two folders which are 'train' and 'test'. Both of these folders contain cracked images along with non-cracked images. There are in total of 227 test images that are used in the following experiment. Python is the choice of the programming language used for this experiment along with sci-kit learn as the library used for the process of classification. Initially, the Python program is written in Jupyter Notebook. Various libraries have been imported for making sure the SVM is implemented correctly and satisfactorily.

```
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
%matplotlib notebook
from sklearn import svm, metrics, datasets
from sklearn.utils import Bunch
from sklearn.model_selection import GridSearchCV, train_test_split
from skimage.io import imread
from skimage.transform import resize
```

Above are all the libraries that have been imported to implement this support vector machine model. matplotlib is a library used to plot and it also provides an object-oriented API for embedding these plots. numpy is the library required for doing any numeric calculations, containing multidimensional arrays and matrix data structures. sklearn also called sci-kit learn is the core library in this experiment. Its functionality includes linear and logistic regression along with classification for which it predominantly uses the kth nearest neighbor algorithm.

```
 def load_image_files(container_path, dimension=(64, 64)):
```

Above is the data loading function, which takes in images files as input and stores them into a container. There are sub-folders within the input folder, where each sub-folder is treated as an individual class.

```
images = []
    flat_data = []
    target = []
    for i, direc in enumerate(folders):
        for file in direc.iterdir():
            if(str(file).endswith(".jpg")):
                img = imread(file)
                img_resized = resize(img, dimension, anti_aliasing=True,
                                                     mode='reflect')
                flat_data.append(img_resized.flatten())
                images.append(img_resized)
                target.append(i)
    flat_data = np.array(flat_data)
    target = np.array(target)
    images = np.array(images)
```

Once the data has been loaded, images and target arrays are created and the folders are enumerated till every '.jpg' has been loaded into the array. Now these array elements are resized and flattened. Once the above processes are performed, the NumPy array contents are copied back to the target and images arrays. Then a bunch is returned which contains the flatten images NumPy array along with the image classification dataset.

Here the split function is not used for splitting the data into train and test sets. Data is already divided into train and test folders. As a split function also does random split, it has been avoided as the dataset has to use a specific number of images as its train and test sets.

```
 param_grid = [
  {'C': [*np.arange(0.1, 1.1, 0.1)], 'kernel': ['linear']},
  {'C': [*np.arange(0.1, 1.1, 0.1)], 'gamma': [0.0001], 'kernel': ['rbf']}
                                         ,
 ]
svc = svm.SVC()
clf = GridSearchCV(svc, param_grid, cv=5, n_jobs=4, verbose=5)
clf.fit(image_dataset_train.data, image_dataset_train.target)
```

The above code snippet shows the parameter optimization used for the experiment. Grid search is also performed specifically for hyperparameter tuning. Cross-validation is also performed (cv=5), which means the model runs 5 times and every time it runs, its accuracy is recorded. After the code runs for 5 times, all the 5 accuracies are summed up and averaged out. The average value obtained would be the cross-validated accuracy which could be considered as the actual accuracy of the model.

```
 y_pred = classifier.predict(image_dataset_test.data)
```

This is the final step in the SVM classifier, where the test dataset is run over the classifier and its testing accuracy is predicted. With the execution of this method, the respective training and testing accuracies are printed out. After the process of cross validation, the SVM classifier was able to generate an accuracy of about 82 percent on the validation set. Below is the screenshot of one of the runs.

5.2 Image Classification using MatConvNet

After the first experiment performed using the SVM classifier, the second experiment involves the usage of deep learning to classify the dataset. Here a variant of VGG-16 has been used.

Table 3: Cross validation results for training and testing accuracy of SVM.

Fold	**Linear SVM Training Accuracy (%)**	**Linear SVM Testing Accuracy(%)**
1	100	85.84
2	100	84.07
3	100	85.40
4	100	75.66
5	100	79.20
Average	**100**	**82.03**

This deep learning neural network consists of 4 convolutional layers L1, L3, L5, and L7, two pooling layers L2 and L4, a ReLU layer L6 and the final softmax layer L8. Note that batch normalization happens at all the convolutional layers. The same data that was used in the SVM classifier has been used for training and validation. Here for the training, we use 2,028 images and for validation, we use 227 images. Out of the entire dataset, 10 percent of the images are used for the process of validation.

```
import numpy as np
import os
import pickle
import tensorflow as tf
import time
from datetime import timedelta
import cv2,sys
from pathlib import Path
```

As observed above, Tensorflow is imported, as it serves as our primary deep learning framework throughout this experiment. Along with that, we utilize the basic NumPy for numerical calculations and also for the creation of NumPy arrays. DateTime and time delta are used to keep track of how long the model takes to train the network completely. It might be surprising to see a pickle library imported in the above code snippet, the reason being creating a '.pkl' cache file. Before starting the training process, make sure to cache the input data into a '.pkl' file for usage throughout the entirety of the training process. One more advantage of using a pickle file is to save the model into a '.pkl' file as it is less time-consuming.

Once the necessary libraries have been successfully imported, the next step performed here is to specify the input directory from where the images need to load. The entire path of the input directory needs to be provided. Similarly, an output directory also needs to be specified along with its path where the TensorFlow model can be saved. Part of this experiment also includes the creation of a cache-wrapper function. As explained in the previous section, this function is solely responsible for identifying if there is an existing cache file or not. Followed by which, another wrapper function is used for creating a dataset object. The purpose of this is to make sure that the order in which the filenames appear will be consistent every time the data is loaded.

```
def one_hot_encoded(class_numbers, num_classes=None):
```

One hot encoding function is used for the generation of an integer array. This particular function takes in two values which are class numbers and number of classes. The number of classes functionality is unique, where depending on the number of folders, the number of classes is created. For example, in our experiment, we have two folders namely ‘cracked’ and ‘non-cracked’. So here the method iterates over the folders and every folder is treated as a class.

A dataset method is specified which takes in the input data. Depending on the structure of the directory, the code detects how many classes are there based on the previous code snippet where we specified the number of classes. It is necessary to stick to the following directory structure, especially for this experiment as the data has been arranged in a particular way. If the structure of the directory is changed, it is important to make necessary changes to the code which takes in the input data, as directory structures can be specific to the way data may be loaded and classes determined. The order in which the directory has been structured for our dataset is as follows: “Cracky/Crack/Train/Test”. And similarly “Cracky/No-Crack/Train/Test”. So here we have two classes, ‘Crack’ and ‘No-Crack’. The number of sub-folders within the root folder determines the number of classes.

The initialization of various layers of the neural network would be the next few steps that are undertaken. The first convolution layer is initiated which specifies the input (the previous layer), number of input channels, size of the filter which is the width and height of the filter, along with the number of filters.

```
def new_conv_layer(input,num_input_channels,filter_size,num_filters):
```

Max pool layers are initiated with layer details such as layer size, layer ksize and the number of filters in the layer, and strides.

```
def max_pool(layer,ksize,strides):
```

```
def new_fc_layer(input,num_inputs,num_outputs,use_relu=True):
```

Post creation of all layers, the input data is given to the neural network to train for a specific number of iterations. Here 1,500 iterations are used. The individual batch size is 64 images.

```
model.optimize(num_iterations)
```

As opposed to the SVM classifier, this deep learning model produced a staggering 92 percent accuracy. The accuracy was confirmed after a cross-validation procedure where the model is trained several times and averaged out.

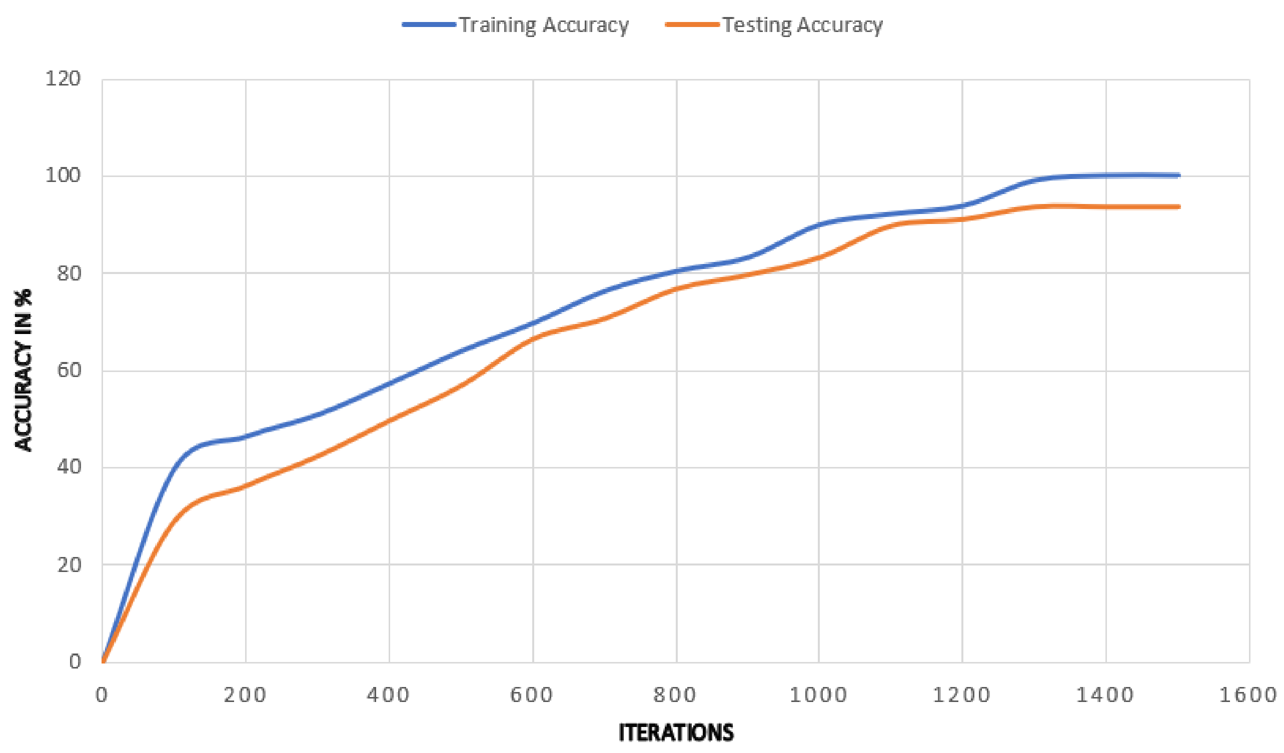


Figure 12: Accuracy of image classification using MatConvNet.

5.3 Image Classification using VGG-16

A critical requirement of these studies is to draw in a comparison between various machine learning models. As explained and shown in the previous section where the data was run over a MatConvNet, similarly the data is run over a VGG-16 network. Here it is to be noted that we have not used transfer learning, but instead have opted to use it as a fully trained network. The VGG-16 model has been built from scratch with a definitive layer set. Here tweaking is done to swiftly be able to run our dataset over the network. Below we can see the libraries imported and necessary to start building this model.

```
import keras, os
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPool2D, Flatten
from keras.preprocessing.image import ImageDataGenerator
import numpy as np
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint, EarlyStopping
import matplotlib.pyplot as plt
```

Instead of using TensorFlow, we use a TensorFlow wrapper Keras as the primary framework. Sequential is used to run the layers in a specific order. Numpy is used for mathematical calculations and array implementations. Here early stopping is utilized to halt the training once the model reaches the required accuracy and does not go beyond that.

```
def vgg16model():
    model = Sequential()
    model.add(Conv2D(input_shape=(128, 128, 3), filters=64, kernel_size=(3
                                             , 3), padding="same", activation=
                                             "relu"))
    model.add(Conv2D(filters=64, kernel_size=(3, 3), padding="same",
                                             activation="relu"))
```

As observed in the above code snippet, there is a calling of sequential function, which dictates how the below-initiated layers are executed in order. Here a convolutional layer is created with input size the same as that of our images. Using 64 filters and ReLU as the activation layer. Similarly, multiple convolutional and max-pooling layers are created. It has to be noted that all of the convolutional layers use the only ReLU as the activation function.

```
model = vgg16model()
opt = Adam(lr=1e-4)
```

As we can see above, one of the very important aspects of any neural network would be the learning rate. This dictates how fast or slow, the model learns from the given input data. The slower the learning rate, the better trained the neural network will be, provided the data is clean enough. Here we are using an 'adam optimizer' for the learning function. Here the learning rate applies to individual iterations.

Post-training, it was seen that the VGG-16 model has a better accuracy at classification. It was observed that just in 5 epochs the VG-16 model was able to achieve an accuracy of 93.3 percent accuracy. The accuracy increased in comparison with the MatConvNet. On a similar dataset that was run over SVM, MatConvNet, and VGG-16, VGG-16 came out as a better model to use for image classification.

In the table below, the complete results of the 3 experiments are given. All these values have been verified using the N-fold cross-validation approach. All the models are trained 5 times and the resultant accuracies have been averaged out. The table also lists the dedicated time taken by individual models to reach maximum training accuracy. The models are not locally run in the machine but have been run over a dedicated server. It is seen that, even though VGG-16 has better accuracy, it takes more time than SVM and MatConvNet to fully train as it has more layers. The second column shows the accuracy achieved by all the models on the validation test set.

Table 4: Model accuracy and time to reach max training accuracy.

Model	**Test Accuracy(%)**	**Training Time(min)**	**Parameters**
SVM	82.03	2.1	16384
MatConvNet	92.38	4.41	677,643
VGG-16	93.31	16.8	65,062,722

6 Conclusion

In this study, various machine learning approaches were tested. These included simple machine learning models such as support vector machines as well as complex deep learning models. To achieve better training models, data was collected carefully and clean high-resolution images were utilized. It was also made sure every image used is of the resolution 128 X 128 pixels to sustain integrity. It must be noted that the two deep learning models were better at classifying the images as either cracked or non-cracked because of choosing the right dataset. In case of a dataset consisting of a high level of imagery with shadows and poor lighting would have caused a drastic dip in the accuracy. From an application point of view, these studies can be scrutinized further before concluding as to which model can be chosen for use as an image processing software for surficial concrete crack detection. However, our study strongly suggests the VGG-16 model is better at classifying. Because a VGG-16 network has solely been devised for applying in these specific scenarios, it is capable of obtaining the best precision. By looking at the results of the 3 models, we can assert that both deep learning networks have a far bigger potential of being used for application of image classification as they have the liberty of utilizing huge amounts of data for training. Meanwhile, these models can be used for any sort of image processing, not limiting to structural engineering alone. These models have a massive potential in being predictive softwares for healthcare industry where the models could be trained using millions of X-ray, CT scan and MRI based imagery and can be something doctors and healthcare professionals could use as a second opinion before calling in the judgment. However, in contemporary times, these models have been applied to domains such as structural engineering and have given birth to automation in fields other than computer science. Further studies need to be done on various other models with different layers which can be a cornerstone for capturing much deeper features, such as embedding the models with drones to get a real-time inspection of building and bridge surfaces.

References

Cha, Y.-J., Choi, W., & Büyüköztürk, O. (2017). Deep learning-based crack damage detection using convolutional neural networks. *Computer-Aided Civil and Infrastructure Engineering*, *32*(5), 361–378.

Dorafshan, S., Thomas, R. J., Coopmans, C., & Maguire, M. (2018). Deep learning neural networks for sUAS-assisted structural inspections: Feasibility and application. In *2018 International Conference on Unmanned Aircraft Systems (ICUAS)* (pp. 874–882).

Dorafshan, S., Thomas, R. J., & Maguire, M. (2019). Benchmarking image processing algorithms for unmanned aerial system-assisted crack detection in concrete structures. *Infrastructures*, *4*(2), 19.

Gareth, J. (2010). *An Introduction to Statistical Learning: with Applications in R*. Springer Verlag.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (pp. 1097–1105).

Moussa, G., & Hussain, K. (2011). A new technique for automatic detection and parameters estimation of pavement crack. In *4th International Multi-Conference on Engineering Technology Innovation (IMETI).*

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *27th International Conference on Machine Learning (ICML)* (pp. 807–814).

Özgenel, Ç. F., & Sorguç, A. G. (2018). Performance comparison of pretrained convolutional neural networks on crack detection in buildings. In *ISARC Proceedings of the International Symposium on Automation and Robotics in Construction* (pp. 1–8).

Scherer, D., Müller, A., & Behnke, S. (2010). Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networks* (pp. 92–101).

Zhang, L., Yang, F., Zhang, Y. D., & Zhu, Y. J. (2016). Road crack detection using deep convolutional neural network. In *2016 IEEE International Conference on Image Processing (ICIP)* (pp. 3708–3712).