Image-based Pothole Detection

Team 9

Qiao Liu

Dandan Zhao

Ching-Min Hu

Matthew Guzman

Table of Contents

Abstract

Pothole detection is a major part in smart city solutions. With efficient detecting techniques, city governments can reduce cost and increase efficiency in road repair operations. In this paper, we tackle the issue of pothole detection in urban areas with image-based machine learning methods. We compared the performance of several models and established a model that can accurately detect potholes on city streets.

1. Project Description

Potholes on roads plague cities and their local governments and drivers. As reported by the American Automobile Association (AAA) in 2016, close to 30 million drivers need to repair the vehicles after encountering a pothole incident. Damages range from tire punctures, bent wheels, suspension damage, exhaust systems damage, and many more. The cost cost of bad roads to American business between now and 2022 is estimated to be $240 billion[1]. City governments have noticed the issue for many years, but cannot fully solve the problem due to inadequate funding and labor.

In recent years, with the rise of Smart City Solutions in the technology industry, computer driven solutions have been proposed for solving the problem. CarVi, a Silicon Valley–based company, expects to share the compiled results from their windshield-mounted pothole detection device with local governments to help locate potholes[2]. The city of Houston also has a web-based service for people to report and locate potholes.

In this study, we propose our own data-driven method to detect potholes on city streets. In this project, we collaborated with the SJSU Smart City Project Team to develop an image-based pothole detection algorithm. The goal is to successfully classify whether an image contains a pothole or no potholes. The dataset included images featuring objects and characteristics of roadways. Objects include manholes on roadways, lane markings, street signs, cars, street signs, buildings and more. Our image detection model strategy aims for versatility towards processing newer information and will include a robust algorithm at the time of deployment. We tested five algorithms: Support Vector Machines (SVM), Logistic Regression, Convolutional Neural Network (CNN), fine-tuned MobileNet model, and a fine-tuned VGG16 model. We will compare these results and establish a system that could be used as a pothole detection service in the future.

2. Requirements

Data preparation: positive and negative datasets that distinguishes potholes with other confusing objects such as manholes.

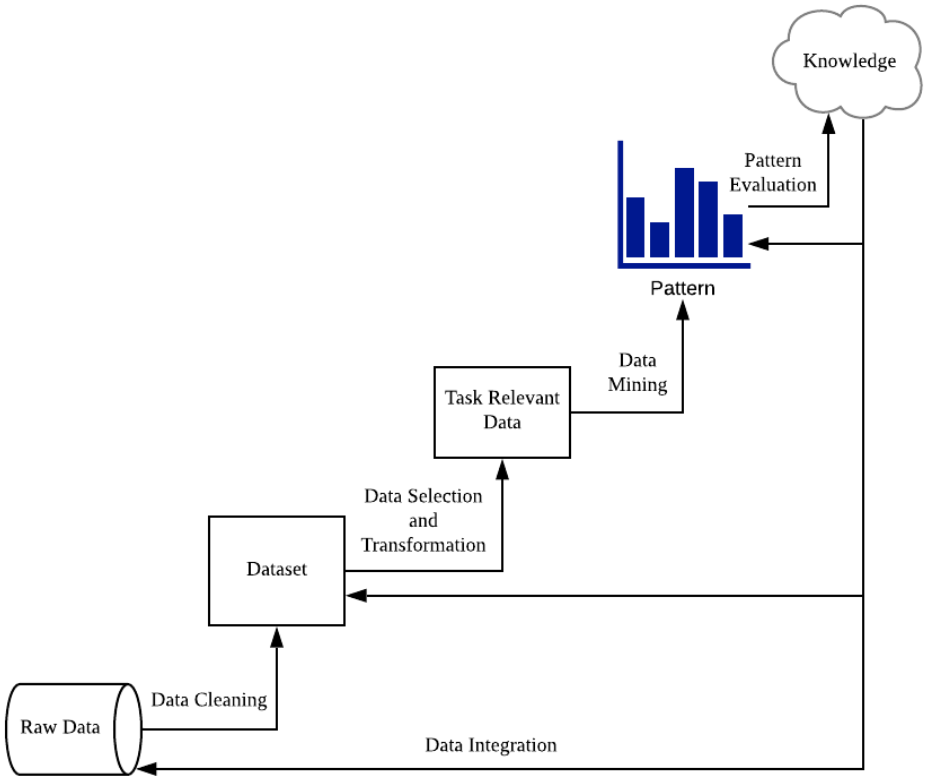
Technical requirement: Platforms for running object detection algorithms, platforms and tools for running UI design.

Testing requirement: datasets that covers different aspects of errors for cornet testing.

As an end result, the model should be robust, and computation time should be in acceptable range. A UI should be designed for the purpose of public access.

Model should have at least a relatively high accuracy that could be acceptable for real-life purpose.

3. KDD



We follow strictly the KDD procedure. We did several trials and revised both the dataset and algorithm to achieve the best possible results. We initially established a training dataset with images from both the internet and self-took photos and trained it with a few basic machine learning

models such as SVM and Logistic Results. After examining the results, we determine that these models were too simple for our purpose, so we turn towards pre-trained CNN models.

After the first round of training with CNN model, we discovered that the result did not meet our requirements. We examined performance reports such as confusion matrix and loss curve, the model did not exhibit extreme deviation from our requirements. The model did not show extreme underfitting pattern.

So we go back to the beginning and increased the dataset, also added some confusing features such as manholes and shadows. After re-training the model on the new dataset, model accuracy greatly improved.

4. Feature Engineering

After careful studying the dataset and a few trials, we performed the following feature engineering procedure:

For raw data collection:

1. We deleted pictures that does not exhibit clear appearance of potholes. We also selected pictures with different aspects of potholes so our Machine Learning model would recognize potholes under different circumstances.
2. We also added negative data so our trained model would be more robust. After examining original photos of potholes, we determined that manholes and round-shaped shadows are both confusing elements that were commonly seen in a pothole image. So, we added these data into our training set.

For model training:

We utilized Keras ImageDataGenerator[3] class to preprocess our image, so the CNN model could better extract features from the picture. The preprocessing procedure on the images we took were as follows:

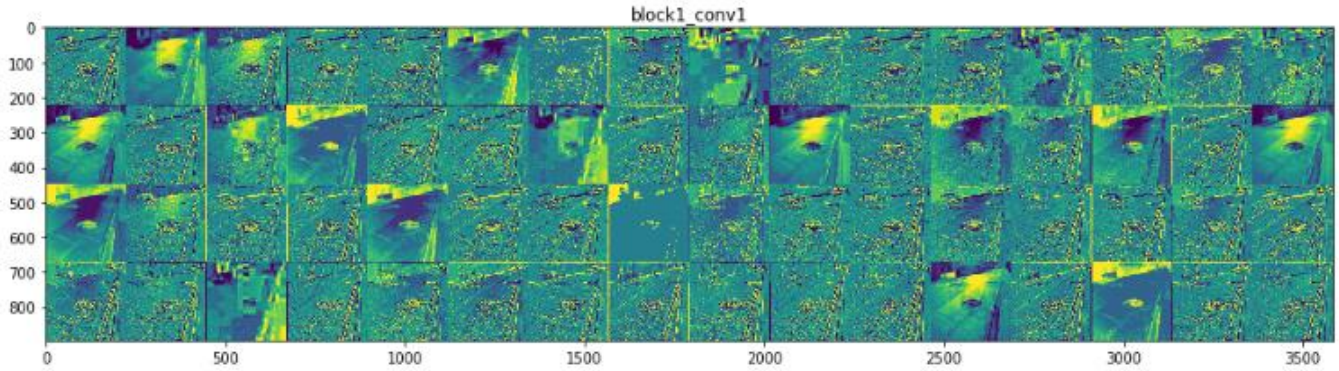
1. Normalized the image data by dividing the pixel matrix by 255.
2. Then we sheared the image within the range of 0.2 degrees to accommodate the fact that our images were taken in different angles.
3. We set a zoom range of 0.2 so the image would be randomly zoomed in the range of 20%. This is for the fact that the pothole in our image is of different sizes, so we try to capture potholes as much as possible.
4. We also randomly flip the images horizontally so that our model would capture potholes of more versatile shape and leaning.

CNN also performance feature extraction when training the data. The process is thoroughly explained in **Data Science Algorithms & Features Used** section. For demonstration, we will give an example below.

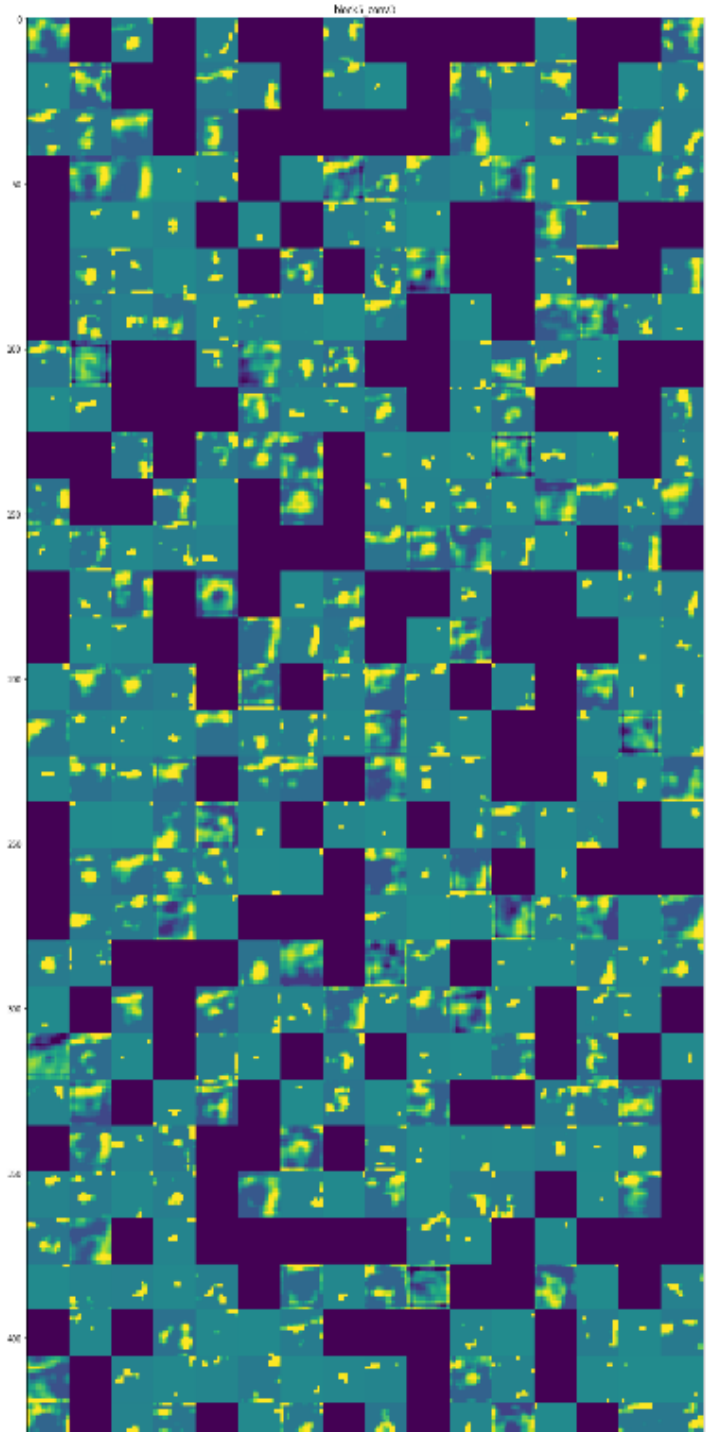
This is the initial image before inputting into our VGG-16 model.

```
<matplotlib.image.AxesImage at 0x28a5e13e1d0>
```

After the first block of convolutional neural networks, we can see that the data still retains almost all of its features.

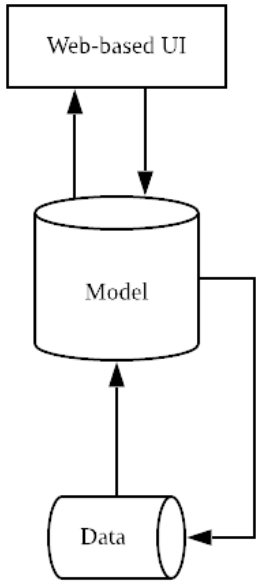


By the last block, we can see that most irrelevant features have been filtered out, leaving only features of the pothole.

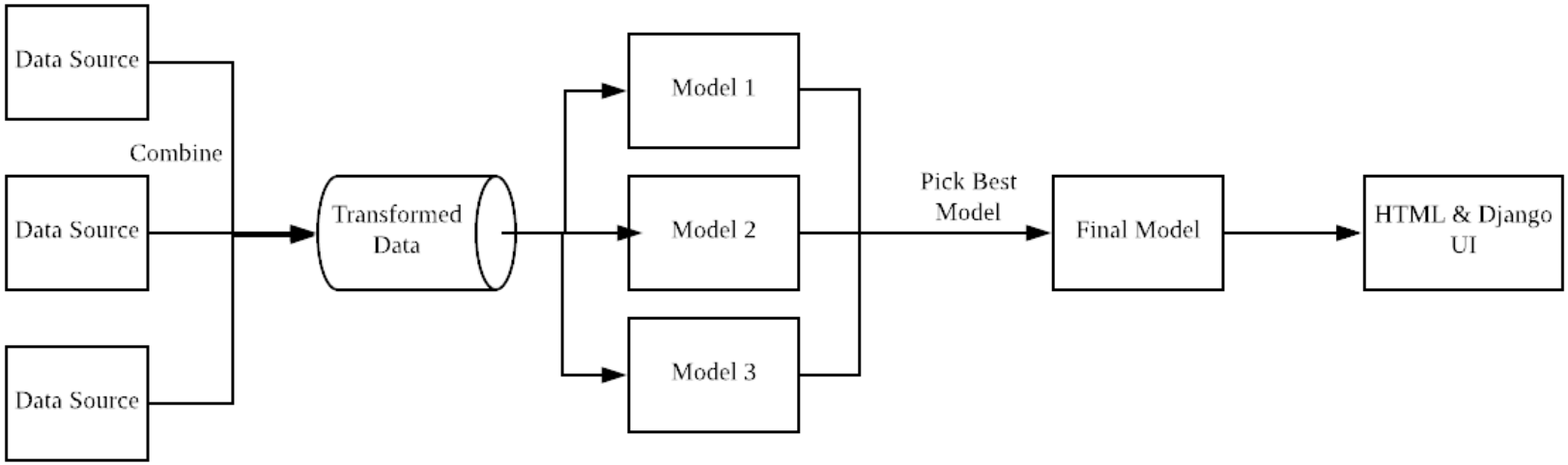


5. High Level Architecture Design

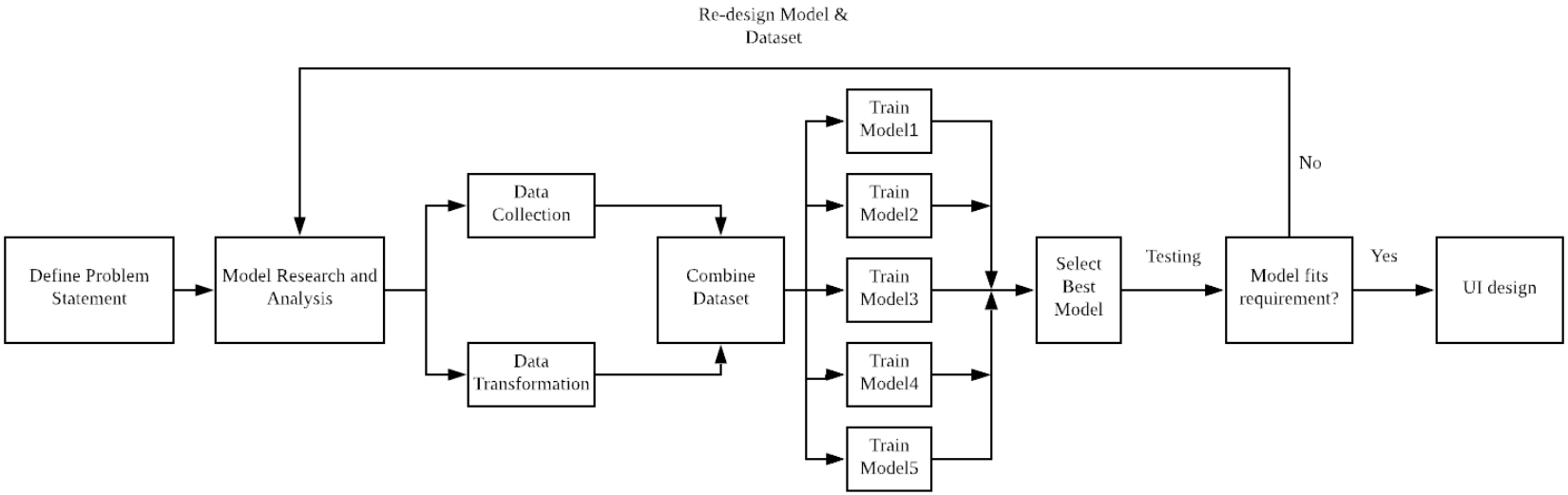
Our system has a single structure, with a backend model running and a UI to accept user input.



6. Data Flow Diagram & Component Level Design



7. Sequence or Workflow

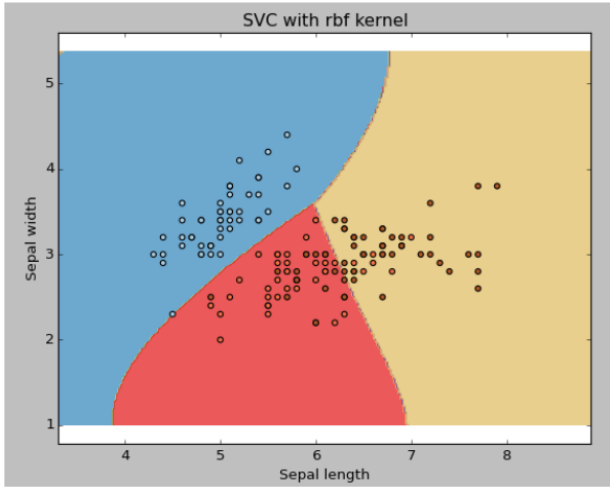


8. Data Science Algorithms & Features Used

1. Support-vector machine (SVM)

SVM[4] is a supervised machine learning algorithm which can be used for both classification and regression methods. With a set of training data, each data is marked as a category, then the model can assign the test data to one of the categories.

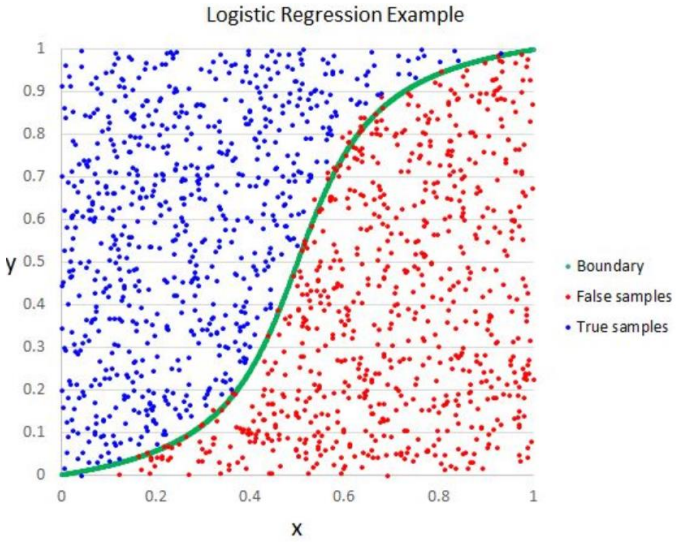
For our project, we take a binary category of pothole or not pothole. For hyperparameter tuning, we chose the ‘rbf’ kernel. This kernel is more flexible than the others, so it fits the purpose for a simple test. Before training, we preprocess the data by labeling the image with pothole or not pothole using labeling tools(LabelMe or LabelImg) then crop the labeled part.



picture source [4]

2. Logistic Regression

Logistic regression[5] uses statistic method and logistic function to model binary dependent variable. The algorithm computes the probability of the two categories (pothole or not pothole). We conducted same preprocessing procedure as the SVM part above.

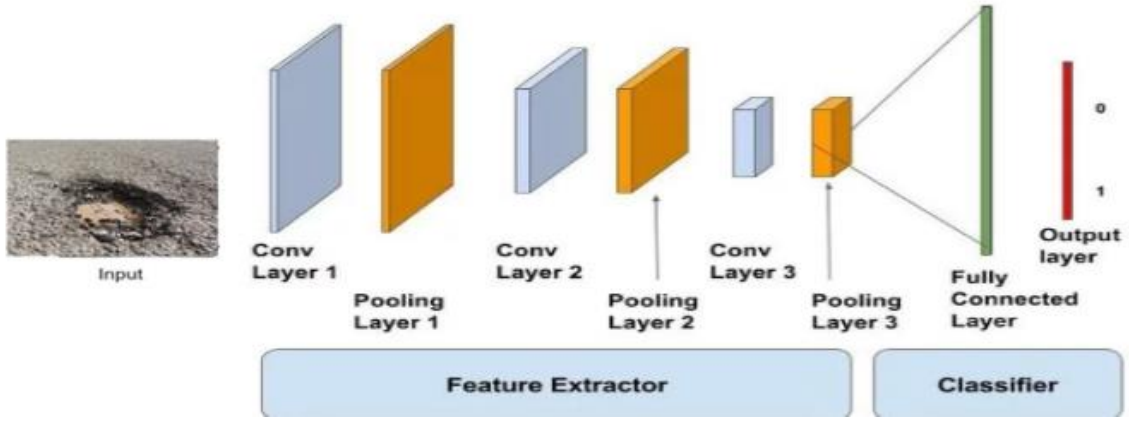


picture source [5]

3. CNN model

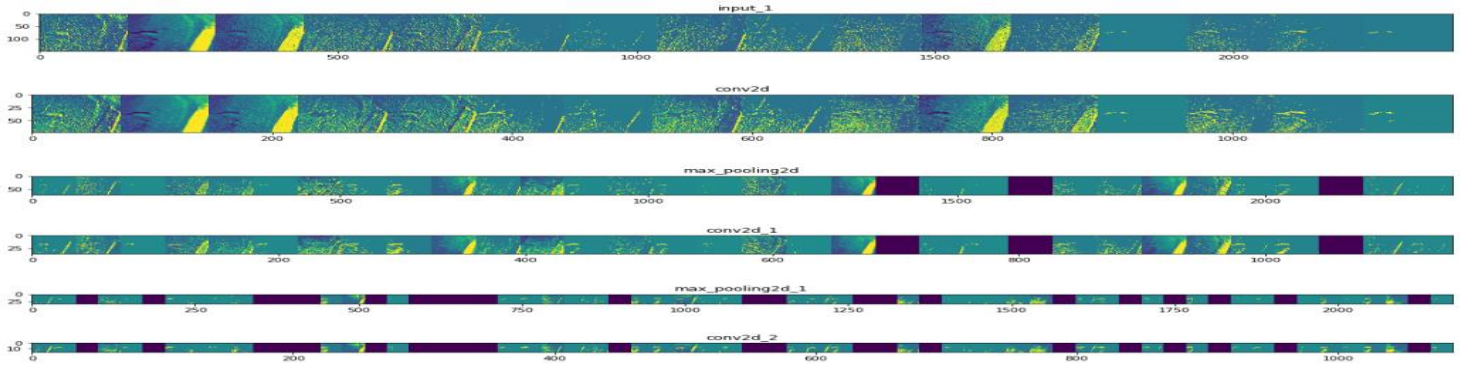
In recent years, Convolutional Neural Networks (CNN) gained much success in image recognition, image classification, and object detection[6]. So we decide to use CNN for our project.

We first built a CNN model from scratch using Keras. The CNN model consist of three convolutional layers, after each layer applied Rectified Linear Unit (ReLU) function and coupled with max-pooling layer as the feature extractor, then feed the features to the fully connected dense layer to do the classification, in order to examine the image is a pothole or not a pothole. The architecture of our CNN model shows below:



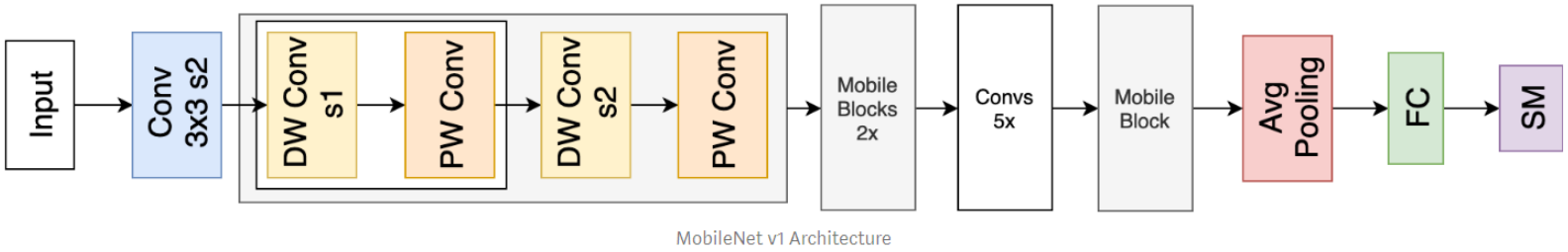
picture source[6]

A Convolution Neural Network (CNN) can extract higher-level representations of images in different layers progressively. With this advantage, we did not need to manually preprocess our datasets, the model itself will learn to extract features with its in-built functions. We built the representations of feature map for each layer, which shows how an image was featured at all levels of CNN layers, the visualization is shown below:



4. Fine-tuned MobileNet model

By using transfer learning, we fine-tuned the MobileNet model. The MobileNet is a pre-trained model which builds on the ImageNet database. MobileNet has the advantage of lightweight, useful for mobile use and embedded vision applications[7]. This characteristic fits our future purpose of training an algorithm for portable devices such as windshield cameras. Here is the architecture of the Mobilenet model:



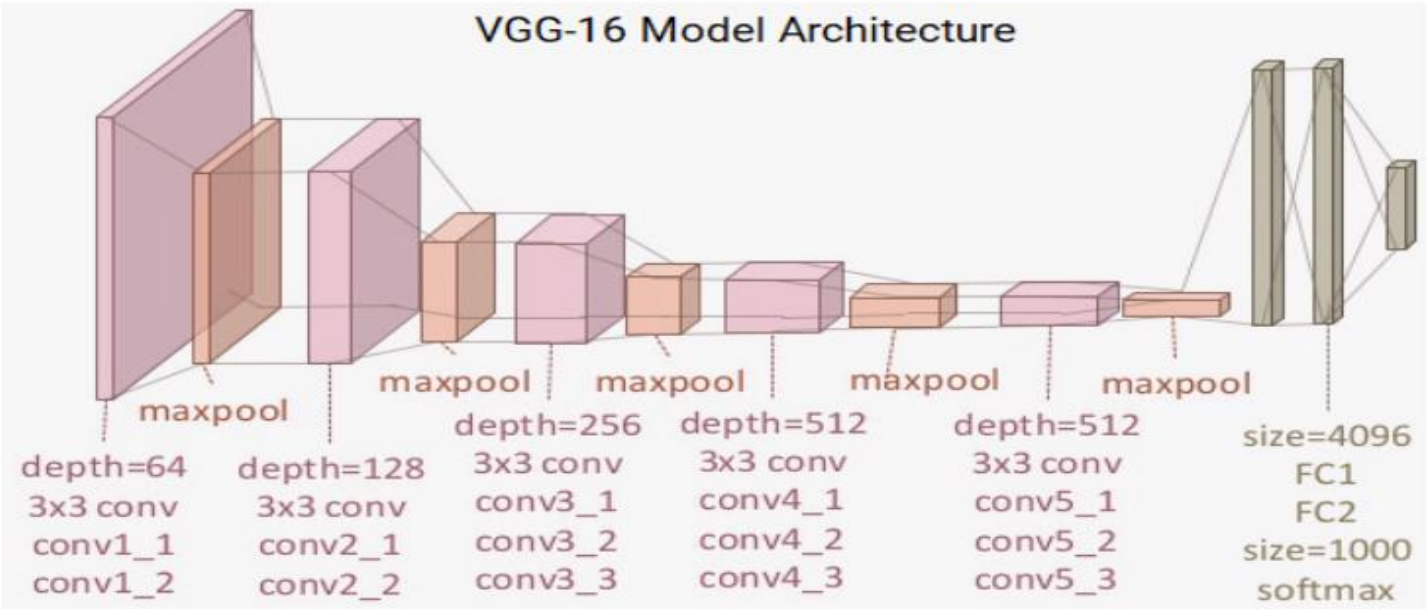
picture source[7]

After initial training, we fine-tuned the model by removing the last three layers and replaced them with our own classifier. The custom layers consisted of three fully connected dense layers and three dropout layers (0.3) to predict the image. Then we retrained the last two blocks of the model, from 74 layer to 94 layer, and the weights on these layers got updated due to the backpropagation operation during each epoch, we could get the performance of how the model worked on our pothole dataset. The part of the detail of the layers show as below (there were 94 layers so we cannot cover all in one screenshot):

70	<keras.layers.advanced_activations.ReLU object...	conv_dw_11_relu	False
71	<keras.layers.convolutional.Conv2D object at 0...	conv_pw_11	False
72	<keras.layers.normalization.BatchNormalization...	conv_pw_11_bn	False
73	<keras.layers.advanced_activations.ReLU object...	conv_pw_11_relu	False
74	<keras.layers.convolutional.ZeroPadding2D obje...	conv_pad_12	True
75	<keras.layers.convolutional.DepthwiseConv2D ob...	conv_dw_12	True
76	<keras.layers.normalization.BatchNormalization...	conv_dw_12_bn	True
77	<keras.layers.advanced_activations.ReLU object...	conv_dw_12_relu	True
78	<keras.layers.convolutional.Conv2D object at 0...	conv_pw_12	True
79	<keras.layers.normalization.BatchNormalization...	conv_pw_12_bn	True
80	<keras.layers.advanced_activations.ReLU object...	conv_pw_12_relu	True
81	<keras.layers.convolutional.DepthwiseConv2D ob...	conv_dw_13	True
82	<keras.layers.normalization.BatchNormalization...	conv_dw_13_bn	True
83	<keras.layers.advanced_activations.ReLU object...	conv_dw_13_relu	True
84	<keras.layers.convolutional.Conv2D object at 0...	conv_pw_13	True
85	<keras.layers.normalization.BatchNormalization...	conv_pw_13_bn	True
86	<keras.layers.advanced_activations.ReLU object...	conv_pw_13_relu	True
87	<keras.layers.pooling.GlobalAveragePooling2D o...	global_average_pooling2d_1	True
88	<keras.layers.core.Dense object at 0x0000025D5...	dense_1	True
89	<keras.layers.core.Dropout object at 0x0000025...	dropout_1	True
90	<keras.layers.core.Dense object at 0x0000025D5...	dense_2	True
91	<keras.layers.core.Dropout object at 0x0000025...	dropout_2	True
92	<keras.layers.core.Dense object at 0x0000025D5...	dense_3	True
93	<keras.layers.core.Dropout object at 0x0000025...	dropout_3	True
94	<keras.layers.core.Dense object at 0x0000025D5...	dense_4	True

5. Fine_tuned VGG16 model

Additionally, we also fine-tuned the VGG16 model. The VGG16 model had 16 layers built on the ImageNet database. The VGG16 model is one of the famous models submitted to ILSVRC-2014, which achieves top-5 test accuracy in ImageNet[8]. So, we were very interested in how the VGG16 model worked on our pothole dataset. The following is the architecture of the VGG16 model:



VGG-16 Model Architecture

picture source[8]

As explained before, we were most interested in how the VGG16 model works on our pothole dataset. We removed the last two blocks and replaced them with our own classifier. The custom layers consisted of three fully connected layers and two dropout layers (0.3) which were very similar to what we did on the Mobilenet mobile. Then we unfroze the last two blocks while keeping the first three blocks frozen to retrain the model. The details of the layers show as below:

	Layer Type	Layer Name	Layer Trainable
0	<keras.engine.input_layer.InputLayer object at...	input_1	False
1	<keras.layers.convolutional.Conv2D object at 0...	block1_conv1	False
2	<keras.layers.convolutional.Conv2D object at 0...	block1_conv2	False
3	<keras.layers.pooling.MaxPooling2D object at 0...	block1_pool	False
4	<keras.layers.convolutional.Conv2D object at 0...	block2_conv1	False
5	<keras.layers.convolutional.Conv2D object at 0...	block2_conv2	False
6	<keras.layers.pooling.MaxPooling2D object at 0...	block2_pool	False
7	<keras.layers.convolutional.Conv2D object at 0...	block3_conv1	False
8	<keras.layers.convolutional.Conv2D object at 0...	block3_conv2	False
9	<keras.layers.convolutional.Conv2D object at 0...	block3_conv3	False
10	<keras.layers.pooling.MaxPooling2D object at 0...	block3_pool	False
11	<keras.layers.convolutional.Conv2D object at 0...	block4_conv1	True
12	<keras.layers.convolutional.Conv2D object at 0...	block4_conv2	True
13	<keras.layers.convolutional.Conv2D object at 0...	block4_conv3	True
14	<keras.layers.pooling.MaxPooling2D object at 0...	block4_pool	True
15	<keras.layers.convolutional.Conv2D object at 0...	block5_conv1	True
16	<keras.layers.convolutional.Conv2D object at 0...	block5_conv2	True
17	<keras.layers.convolutional.Conv2D object at 0...	block5_conv3	True
18	<keras.layers.pooling.MaxPooling2D object at 0...	block5_pool	True
19	<keras.layers.core.Flatten object at 0x000001E...	flatten_1	True

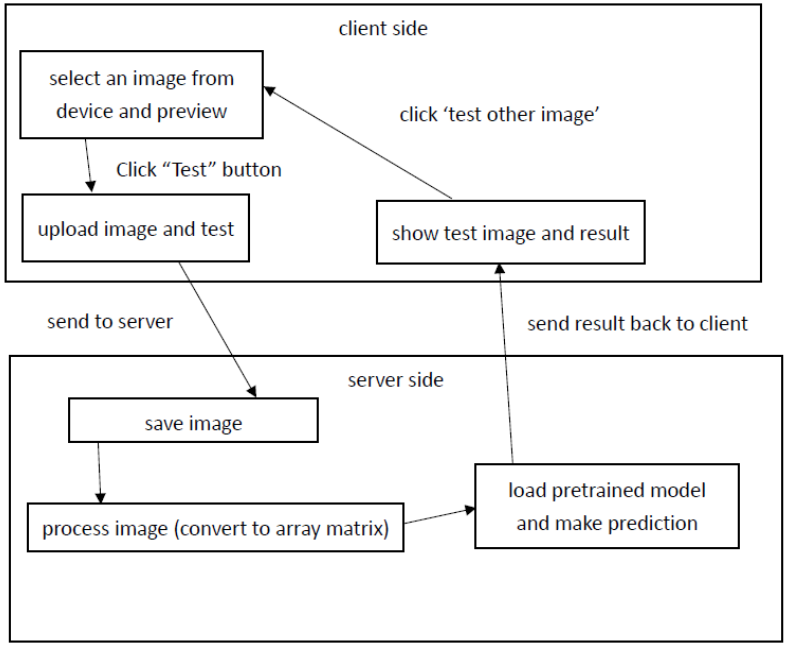
9. Interfaces – RESTFul & Server Side Design

The server side is developed with Python and Django 2.2.6 module[9]. For the server side, it will receive and save the image from client side. After receiving the image, it will transform to matrix first and make prediction by pretrained model -- Fine-tuned VGG16. Then the result of pothole, not pothole result will be sent back to the client side.

10. Client-Side Design

The client side was developed with HTML, JavaScript, and CSS. For the client-side UI, users can upload an image from their device and get the yes/no result on the existence of potholes in that image. The result will be displayed with the image in 320*320 pixels format. When get the result, user can test another image by clicking ‘test other image’ button then client side will redirect to initial page.

The design structure of the application is shown below:



Some requirements of the application:

1) Make sure the version of Django module is 2.2.6. Downward compatibility of Django is not good, the application may fail in another version, so we need to have a fixed version.

2) Make sure the version of Keras is 2.3.1 or more recent version. If the Keras version is less than 2.3.1, it would throw TypeError when running.

3) Use Google Chrome to browse since some CSS type is not supported in other browsers.

To use the client side, we need to open Anaconda prompt and cd to the project directory, then use the command python manage.py runserver. After this procedure, users will be able to access the client side in their browser by going to http://127.0.0.1:8000/polls/ .

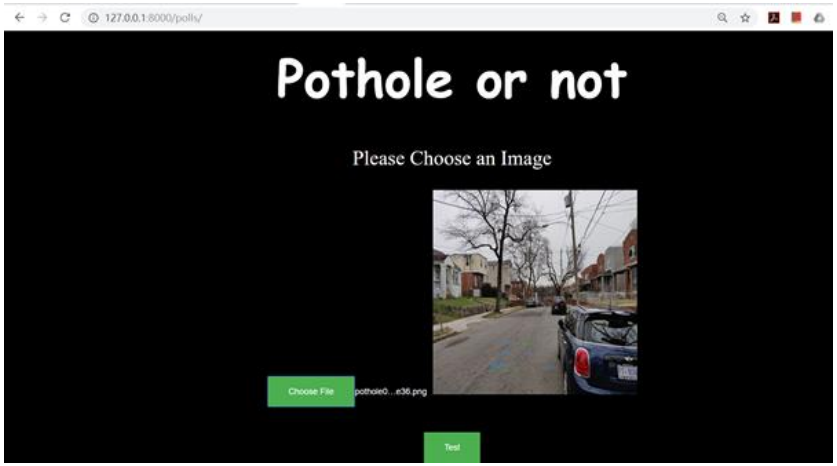
Here are some Demo screenshots of our UI:

Initial page

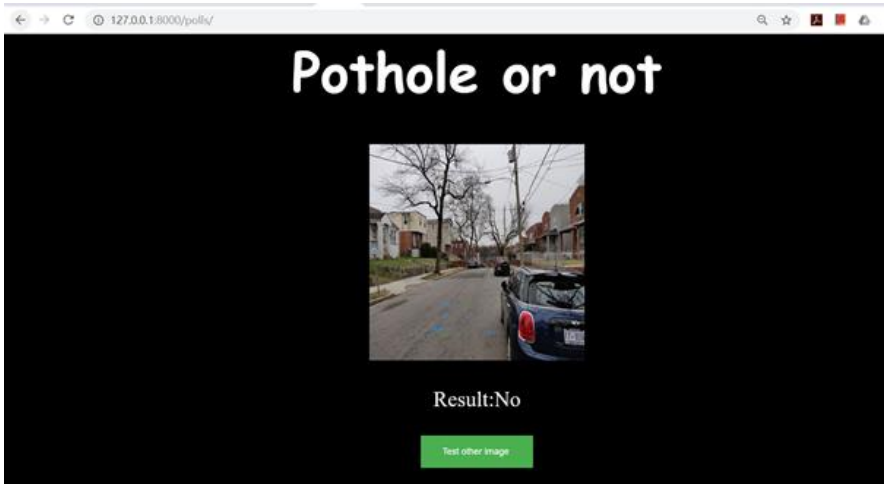
Select an image from device

Result: yes

Test other image



Result: no

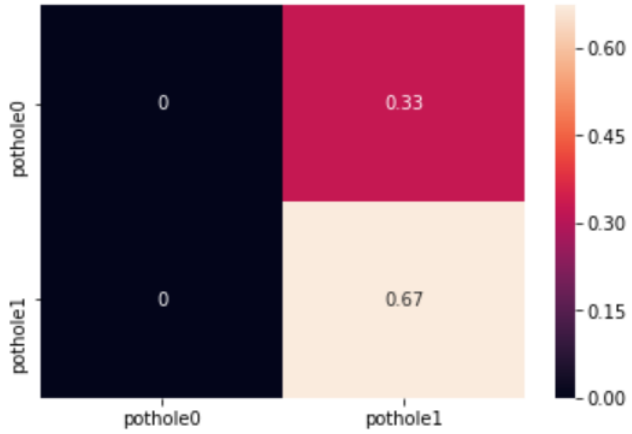


11. Testing (Data Validation /nFold)

1) In SVM model, the image was first resized to 640*640 pixels and decoded in three channels(R,G,B) then convert to gray scale and reshape to one dimension. We chose the ‘rbf’ kernel to train the model with one fold. The size of training data is 1441 (1096 positive and 345 negative) and testing data is 409 (275 positive and 134 negative).

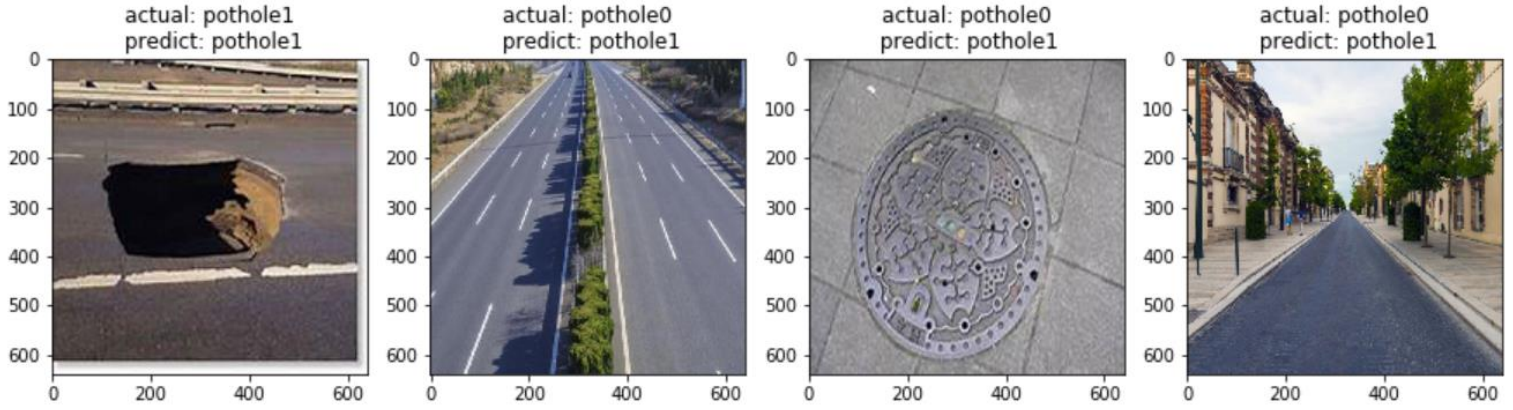
Accuracy score for SVM model was 67.24%

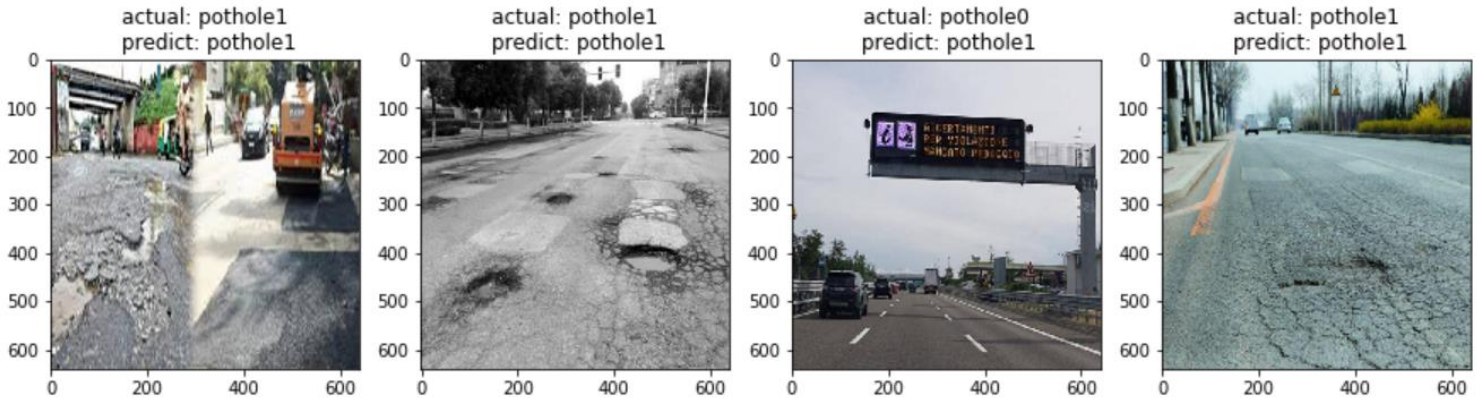
Confusion matrix and classification report of the SVM is shown below. With pothole0 as not pothole and pothole1 as pothole.



	precision	recall	f1-score	support
pothole0	0.00	0.00	0.00	134
pothole1	0.67	1.00	0.80	275
avg / total	0.45	0.67	0.54	409

Example of test data prediction:

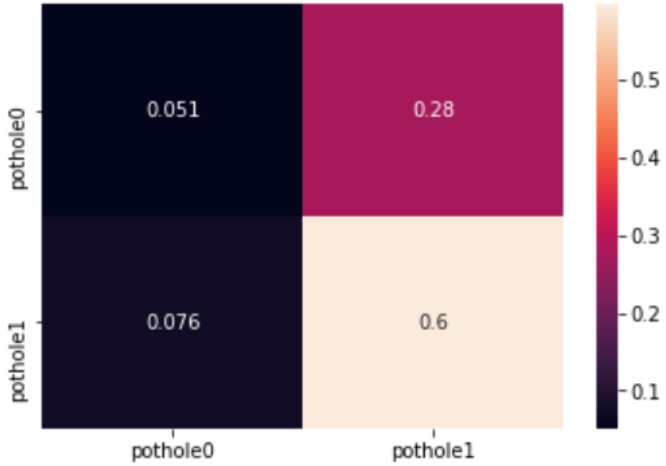




2) In logistic regression, the image was first resized to 640*640 pixels and decoded in three channels(R,G,B) then convert to gray scale and reshape to one dimension. We chose the ‘rbf’ kernel to train the model with one foldThe size of training data is 1441 (1096 positive and 345 negative) and for testing data is 409 (275 positive and 134 negative).

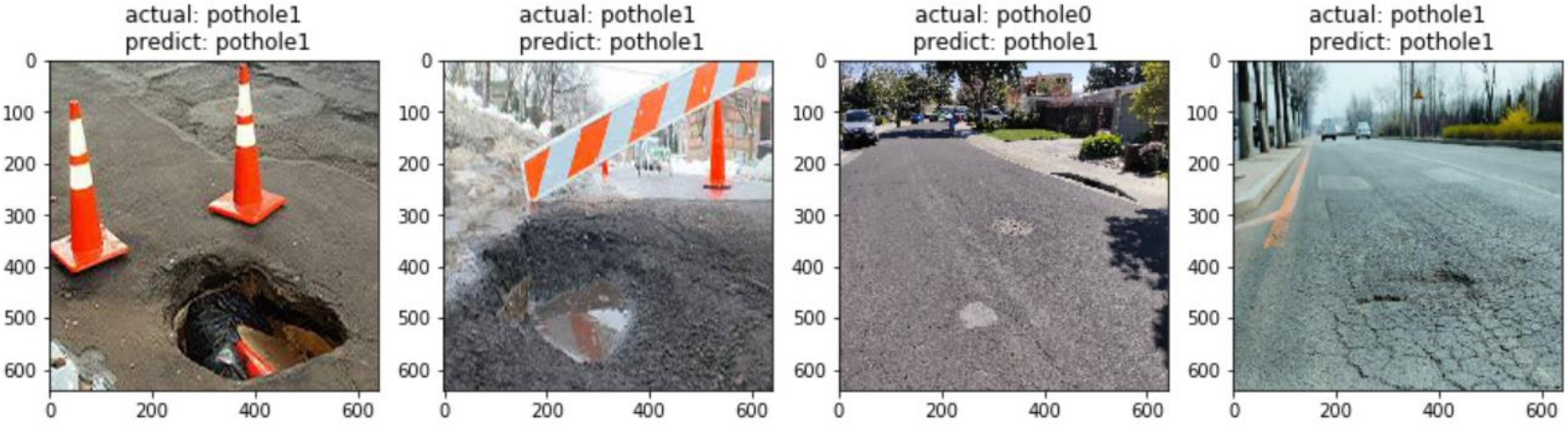
Accuracy score for Logistic Regression model was 64.79%

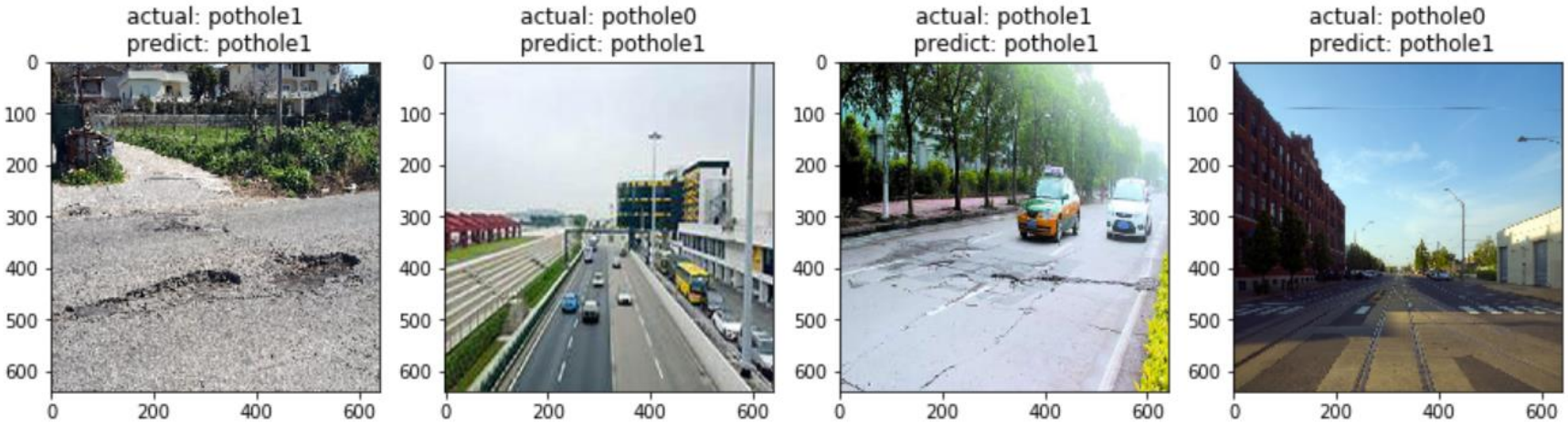
Confusion matrix and classification report of the SVM is shown below. With pothole0 as not pothole and pothole1 as pothole.



	precision	recall	f1-score	support
pothole0	0.40	0.16	0.23	134
pothole1	0.68	0.89	0.77	275
avg / total	0.59	0.65	0.59	409

Example of test data prediction:

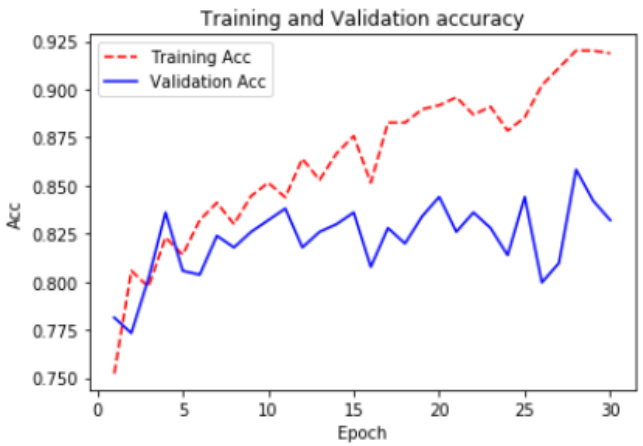




3) For the CNN model, by using Image data generator, we normalized the image by 1./255. The target input shape of data is 150x150x3, with 3 as the three color channels: R, G, and B. The hyperparameters we tuned included: optimizer, learning rate, number of epochs and batch size. We chose Adam optimizer with learning rate 0.001, the batch size was 32 and epoch of 30. The output sample of model shows below:

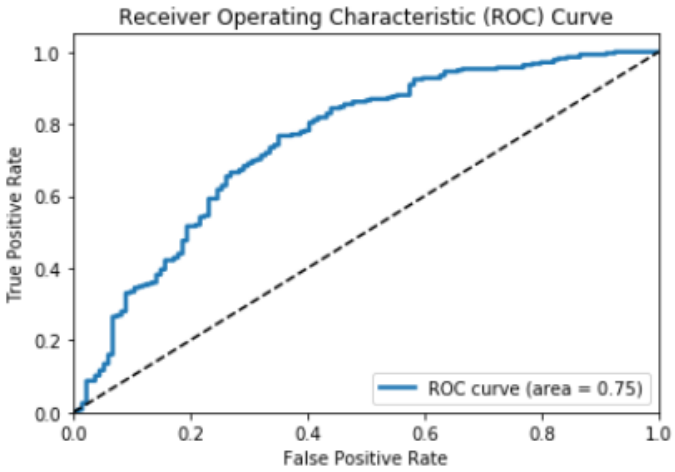
```
Epoch 1/30
16/16 [==============================] - 11s 663ms/step - loss: 0.5637 - acc: 0.7814
46/46 [==============================] - 91s 2s/step - loss: 0.5509 - acc: 0.7523 - val_loss: 0.5637 - val_acc: 0.7
814
Epoch 2/30
16/16 [==============================] - 8s 490ms/step - loss: 0.5671 - acc: 0.7733
46/46 [==============================] - 71s 2s/step - loss: 0.4918 - acc: 0.8057 - val_loss: 0.5671 - val_acc: 0.7
733
Epoch 3/30
16/16 [==============================] - 10s 594ms/step - loss: 0.4632 - acc: 0.8016
46/46 [==============================] - 74s 2s/step - loss: 0.4785 - acc: 0.7974 - val_loss: 0.4632 - val_acc: 0.8
016
Epoch 4/30
16/16 [==============================] - 7s 465ms/step - loss: 0.4625 - acc: 0.8360
46/46 [==============================] - 75s 2s/step - loss: 0.4373 - acc: 0.8230 - val_loss: 0.4625 - val_acc: 0.8
360
```

For the size of training dataset as 1441, the validation dataset as 494 and the test dataset as 275, the accuracy and the loss curve of training data and validation data we got is shown below:



The evaluation of the model on the test dataset is shown below:

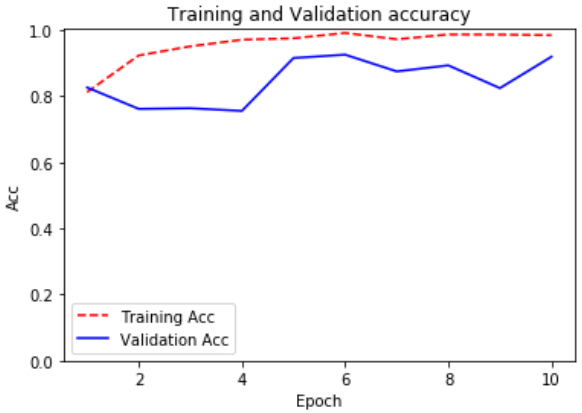
```
Model Performance metrics:
------------------------------
Accuracy: 0.7506
Precision: 0.754
Recall: 0.7506
F1 Score: 0.7178

Model Classification report:
------------------------------
              precision    recall  f1-score   support

    pothole       0.75      0.95      0.84       275
not pothole       0.77      0.34      0.47       134

avg / total       0.75      0.75      0.72       409


Prediction Confusion Matrix:
------------------------------
                      Predicted:
                         pothole not pothole
Actual: pothole              261          14
        not pothole           88          46
```

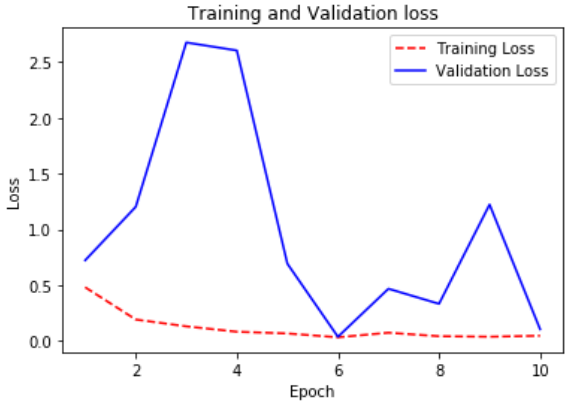


4) For the Fine-tuned Mobilenet model. We used ImageDataGenerator and rescaled the image by 1./255. The shape of our input images is 128 x 128 x 3, with 3 as the three color channels: R, G, and B. The hyperparameters we tuned includes: learning rate, number of epochs and the batch size. We chose batch size of 32 and epoch of 10, the output sample of model is shown below:

```
Epoch 1/10
22/22 [==============================] - 191s 9s/step - loss: 0.4815 - accuracy: 0.8118 - val_loss: 0.7235 - val_accu
racy: 0.8259
Epoch 2/10
22/22 [==============================] - 186s 8s/step - loss: 0.1931 - accuracy: 0.9227 - val_loss: 1.2020 - val_accu
racy: 0.7611
Epoch 3/10
22/22 [==============================] - 220s 10s/step - loss: 0.1319 - accuracy: 0.9503 - val_loss: 2.6735 - val_acc
uracy: 0.7632
Epoch 4/10
22/22 [==============================] - 177s 8s/step - loss: 0.0811 - accuracy: 0.9703 - val_loss: 2.6020 - val_accu
racy: 0.7551
Epoch 5/10
22/22 [==============================] - 179s 8s/step - loss: 0.0686 - accuracy: 0.9746 - val_loss: 0.6929 - val_accu
racy: 0.9150
Epoch 6/10
22/22 [==============================] - 168s 8s/step - loss: 0.0336 - accuracy: 0.9908 - val_loss: 0.0386 - val_accu
racy: 0.9251
Epoch 7/10
22/22 [==============================] - 177s 8s/step - loss: 0.0737 - accuracy: 0.9717 - val_loss: 0.4686 - val_accu
racy: 0.8745
Epoch 8/10
22/22 [==============================] - 187s 8s/step - loss: 0.0432 - accuracy: 0.9862 - val_loss: 0.3335 - val_accu
racy: 0.8927
Epoch 9/10
22/22 [==============================] - 176s 8s/step - loss: 0.0387 - accuracy: 0.9859 - val_loss: 1.2226 - val_accu
racy: 0.8239
Epoch 10/10
22/22 [==============================] - 164s 7s/step - loss: 0.0469 - accuracy: 0.9840 - val_loss: 0.1060 - val_accu
racy: 0.9190
```

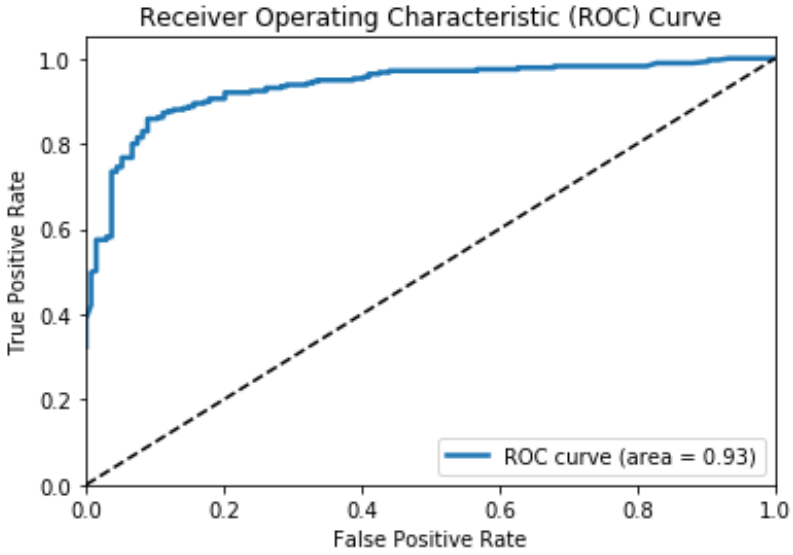
For the size of training data as 1441, the validation data as 494, and the test data as 275, the accuracy and the loss curve of training data and validation data we got as below:





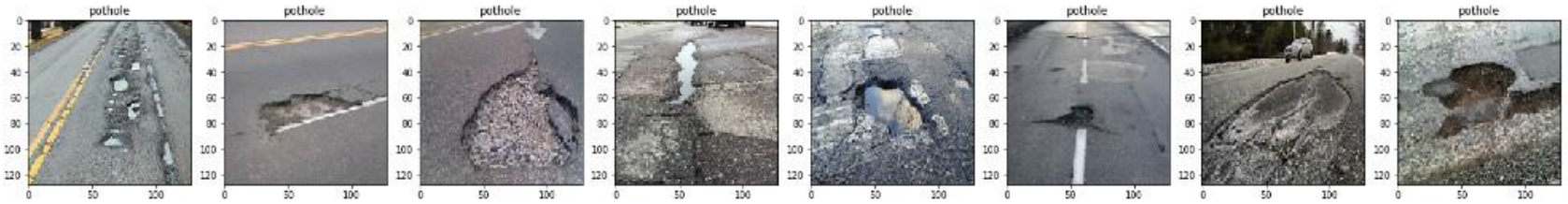
The evaluation of the model on the test dataset is shown below:

```
Model Performance metrics:
------------------------------
Accuracy: 0.8729
Precision: 0.885
Recall: 0.8729
F1 Score: 0.8753

Model Classification report:
------------------------------
             precision    recall  f1-score   support

    pothole       0.95      0.86      0.90       275
not pothole       0.76      0.90      0.82       134

avg / total       0.89      0.87      0.88       409


Prediction Confusion Matrix:
------------------------------
                    Predicted:
                       pothole not pothole
Actual: pothole            236          39
        not pothole         13         121
```

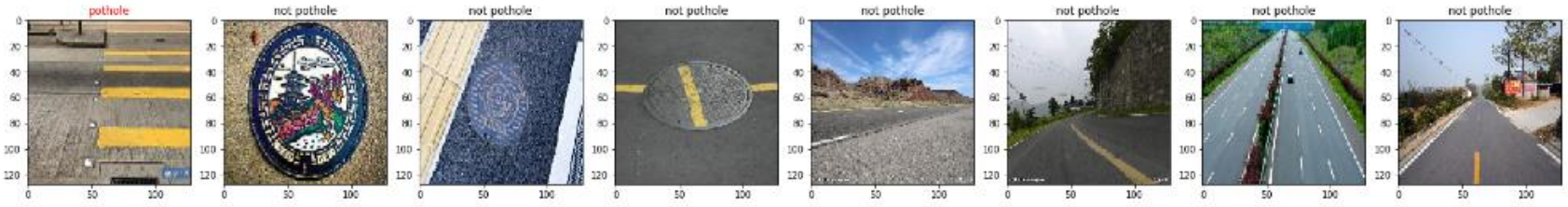


Below are the examples of output about the predictions on the test data by using the fine-tuned Mobilenet model.

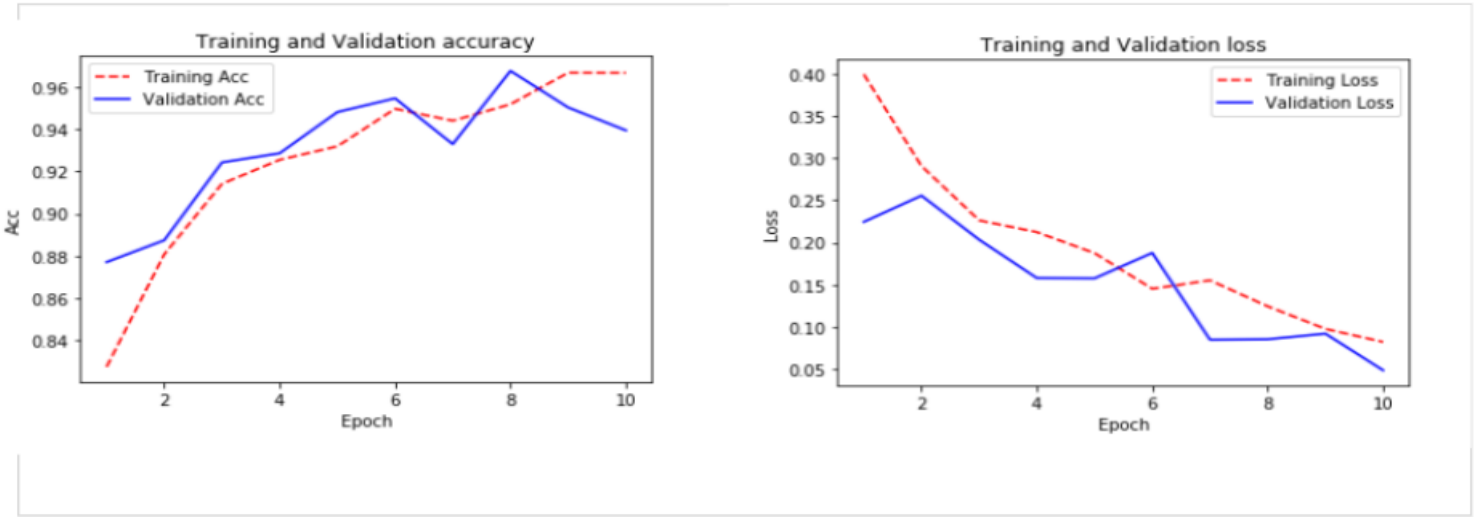
The predictions on the potholes:



The predictions on the not potholes:

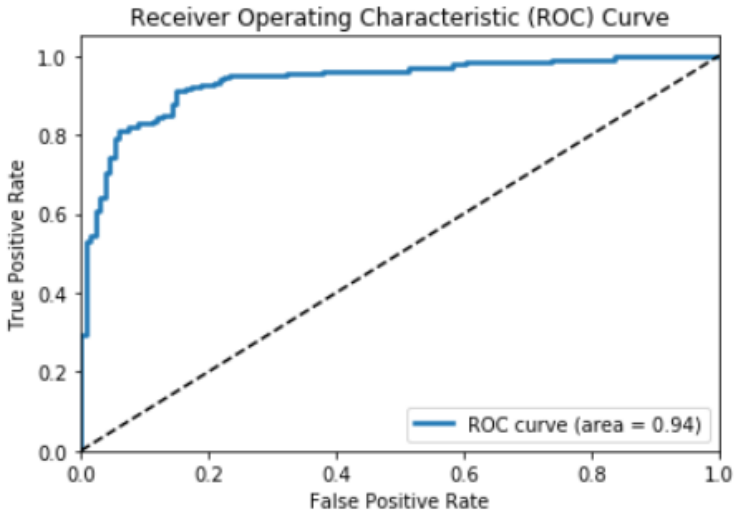


5) For the Fine-tuned VGG16 model, we used ImageDataGenerator to rescale the image by 1./255. Shape of the input image is 128 x 128 x 3 with 3 as the three color channels: R, G, and B. We chose the number of epochs as 10 and the batch size as 32 to retrain the model. For the same size of training data, validation data and test data, the accuracy and the loss curve of training data and validation data we got is shown below:



The evaluation of the fine-tuned VGG16 model on the test dataset:

```
Model Performance metrics:
------------------------------
Accuracy: 0.8509
Precision: 0.8673
Recall: 0.8509
F1 Score: 0.8541

Model Classification report:
------------------------------
             precision    recall  f1-score   support

    pothole       0.94      0.83      0.88       275
not pothole       0.72      0.89      0.80       134

avg / total       0.87      0.85      0.85       409


Prediction Confusion Matrix:
------------------------------
                    Predicted:
                       pothole not pothole
Actual: pothole            229          46
        not pothole         15         119
```



12. Model Deployment

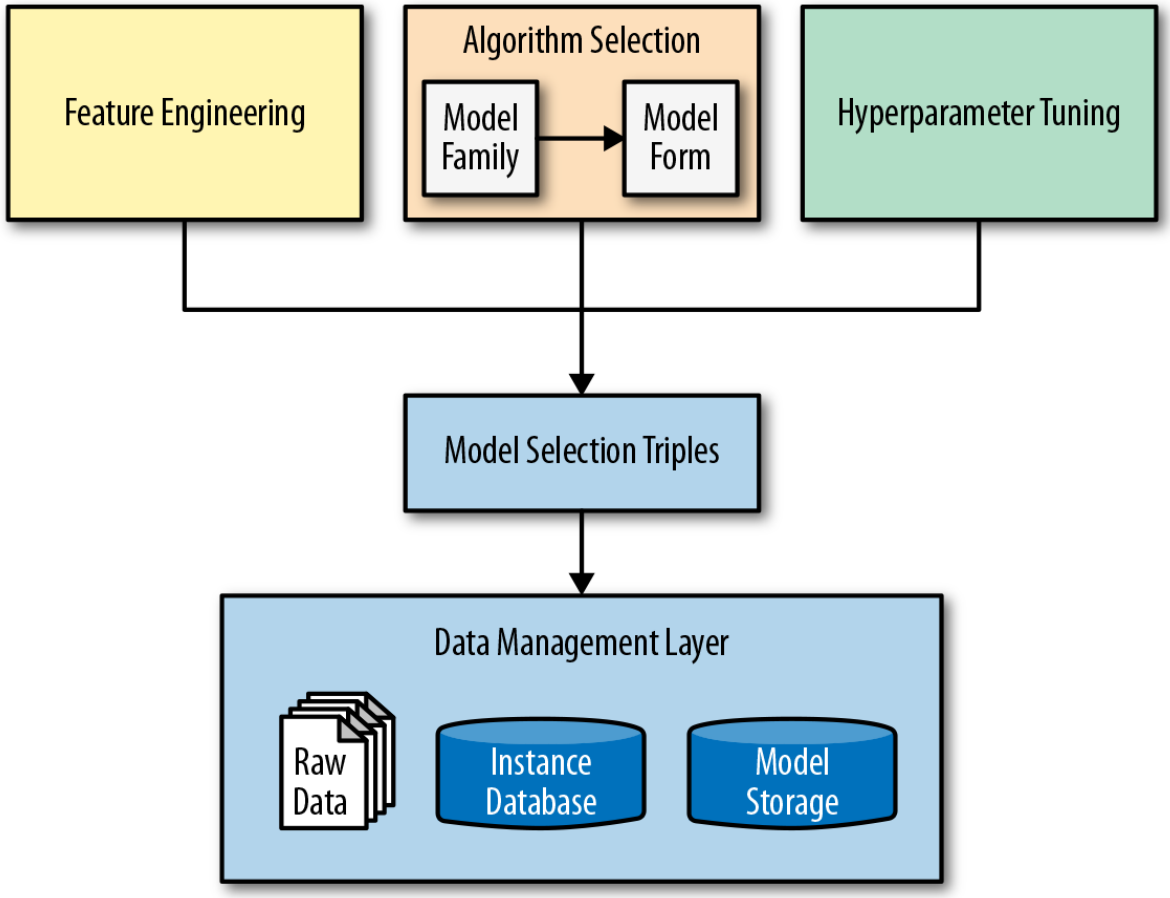
We plan to deploy the application on AWS Elastic Beanstalk or AWS EC2. We have tried to upload the application on AWS Elastic Beanstalk. There's always 'no load balancer', then the deployment cannot finish. With the tutorial which provided in AWS, there's some device environment problems. On AWS EC2, we cannot figure out how to put the application in the virtual machine.

13. HPC

For faster model training speed, we utilized the power of Google Colab GPU run type and Keras tensorflow's built-in parallelization. We tried using PySpark to run our machine learning model in parallelized environment, but Spark's wall time is very long on Google Colab, so we abandoned this try.

14. Design Patterns Used

In this project, we used the standard Machine Learning design pattern for selecting an optimal model. It is an iterative process of cycling through feature engineering, model selection and hyperparameter tuning. During and by the end of each cycle, we evaluate the trained model and compare it with our initial requirement. If the result was not ideal, we will go back and redo the whole process. A diagram of such design pattern is shown below[10]:


Feature Engineering
Algorithm Selection
Model Family
Model Form
Hyperparameter Tuning
Model Selection Triples
Data Management Layer
Raw Data
Instance Database
Model Storage

15. Active Learning or Feedback loop

Building machine learning models is an iterative process. After early research, we aimed at creating a real-time object detection algorithm. We first generated training dataset in XML format and trained a YOLO model. When the results did not show any improvements after several trials and hyperparameter tuning, we decided to search for different models.

We then go back to the standard machine learning process. We first started with simple algorithms such as SVM and Logistic Regression to get a baseline performance of simple models. Accuracy score from simple tests showed that the second version of training data could be trained to get some results, so we will use this form of dataset in later training.

Next, we built a naive CNN model from scratch with Keras. When the results after hyperparameter tuning did not meet our requirements, we turned towards open-source pretrained models.

After carefully studying our problem statement and data type, we decided to use several object detection CNN models such as MobileNet and VGG-16. We first trained our data on the pretrained models, then re-model and fine-tuned hyperparameters based on our understanding of the dataset. The results from our remodified pretrained models exhibited ideal behavior.

With these few loops of model selection process, we gained a better understanding of our dataset and different models along the way.

16. Interpretability of the Model

With research we understood that CNN based models had in-build functions for feature extraction. So the question lies in how to interpret the models, what is the output after each convolutional layer and the max-pooling layer, and what features we get for each layer. By applying visualization techniques[11] we can realize the interpretability of the model. Here we chose our best fine-tuned VGG16 model to explain how the model gets the final predictions.

Similar to the representations we built for each layer of the basic CNN model, we employed the method of feature visualization to visualize the feature map output after each convolution kernel. In other words, by visualizing the result of convolution operations, we understood which features were extracted by the convolutional kernel.

Let's take a look at the summary of our fine-tuned VGG16 model. We have removed the last three full connected dense layer and replaced them with our own classifier of three fully connected dense layers and two dropout layers. The first summary is the VGG16 model after removing the full connected layers and the second part is the VGG16 model followed by our own classifier.

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0

Model: "sequential_1"

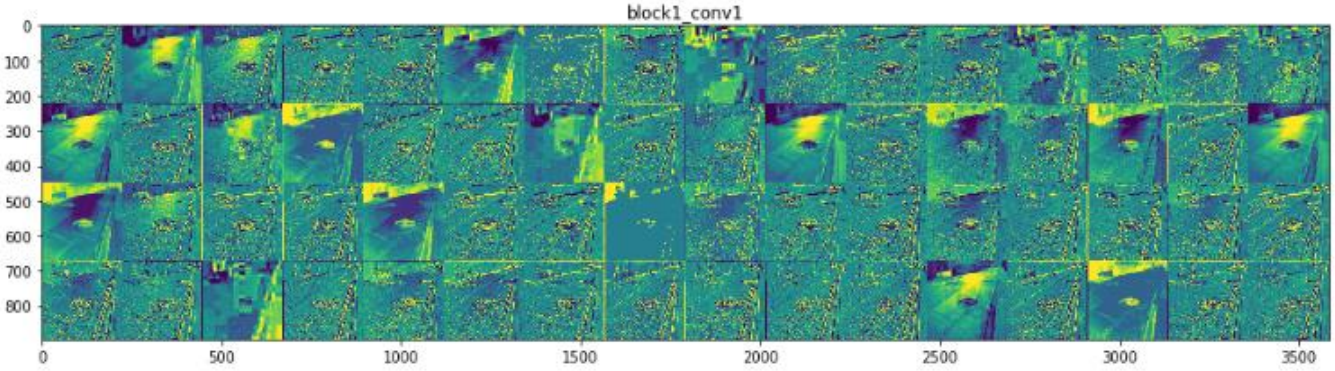
Layer (type)	Output Shape	Param #
model_2 (Model)	(None, 8192)	14714688
dense_1 (Dense)	(None, 512)	4194816
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 512)	262656
dropout_2 (Dropout)	(None, 512)	0
dense_3 (Dense)	(None, 1)	513

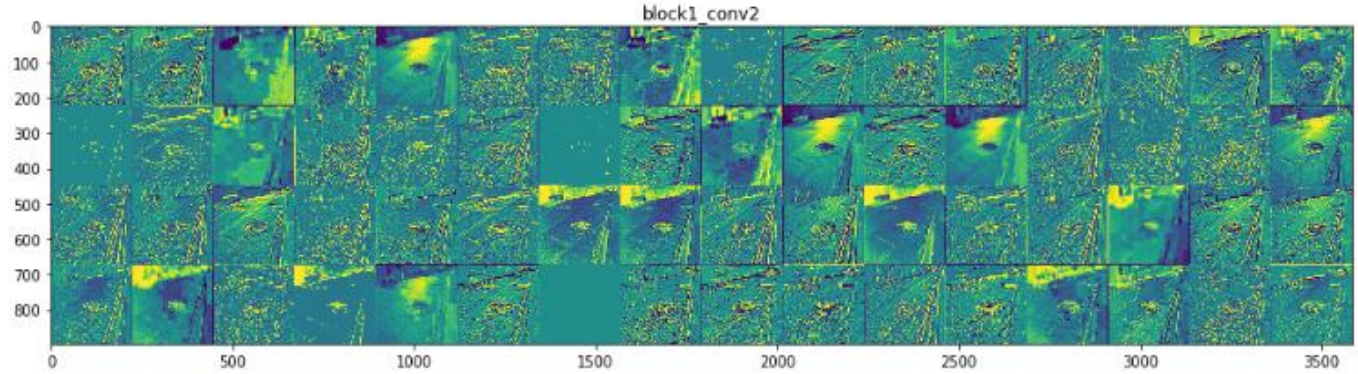
Total params: 19,172,673
Trainable params: 17,437,185
Non-trainable params: 1,735,488

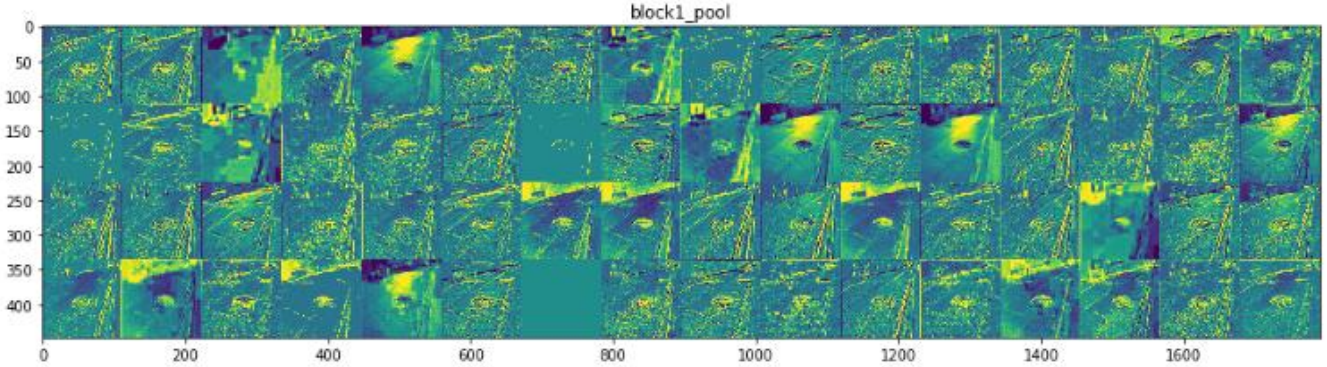
By using feature visualization, given an input image, we can see the features extracted after each conventional layer. There are 3 dimensions: width, height, and the depth (channels) in the feature maps as an input. After the first conventional layer, the output dimension is 64. In other words, we used 64 filters to extract the features after the first conventional layer. So the feature maps are plotted for each channel individually for the fact that each channel encodes relatively independent features.

The following are the initial picture and feature maps generated after each layer:

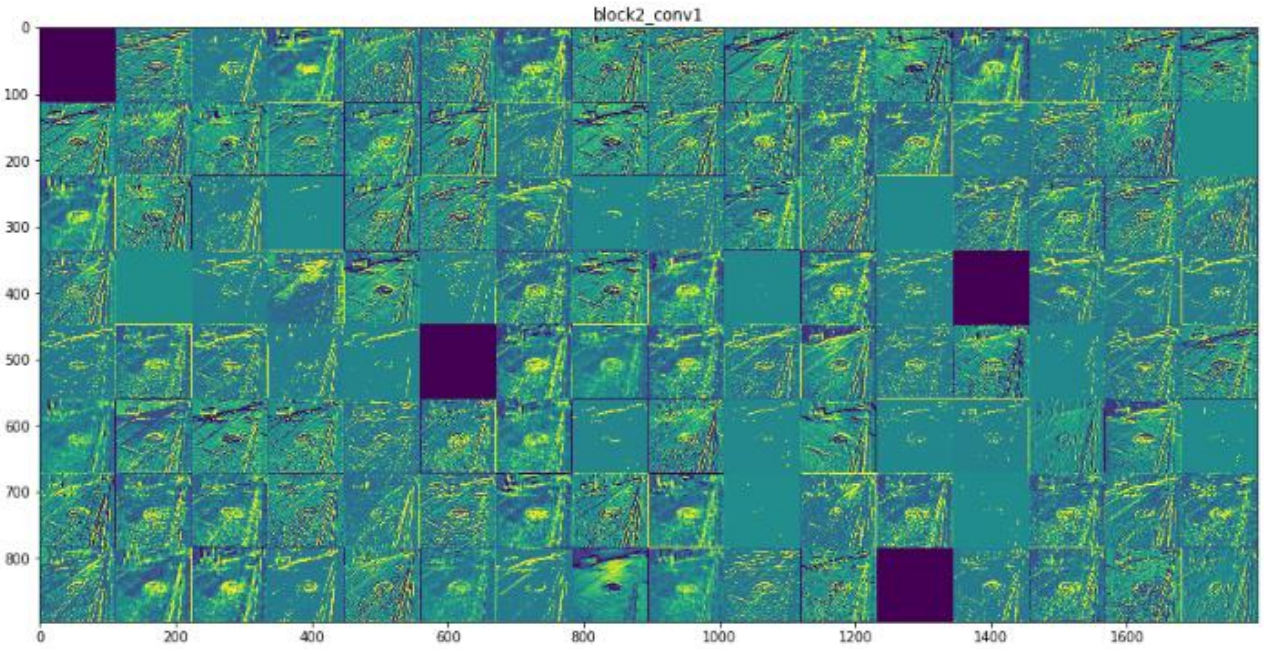
```
<matplotlib.image.AxesImage at 0x28a5e13e1d0>
```

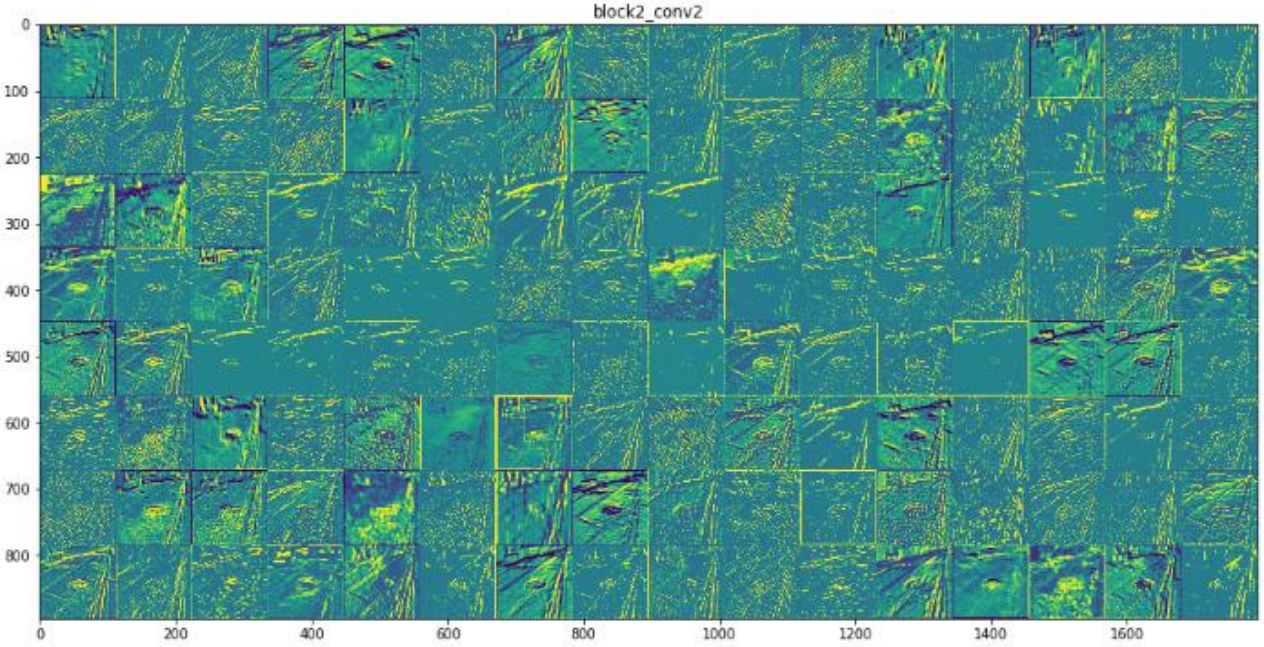


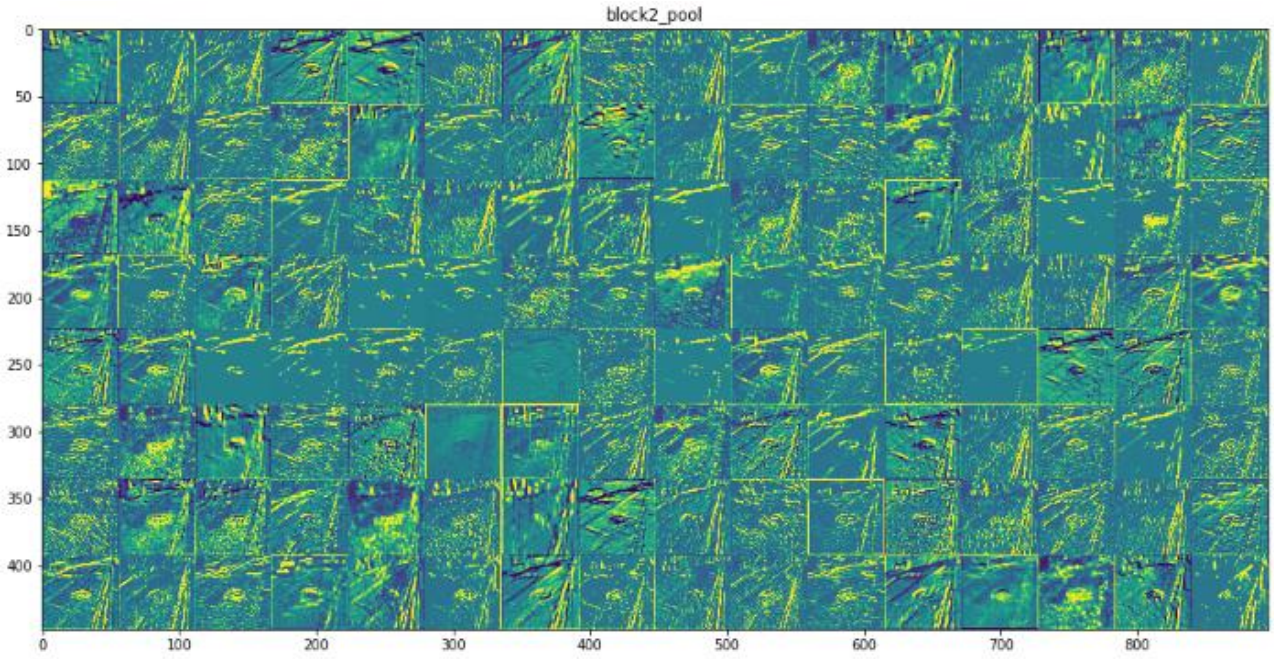




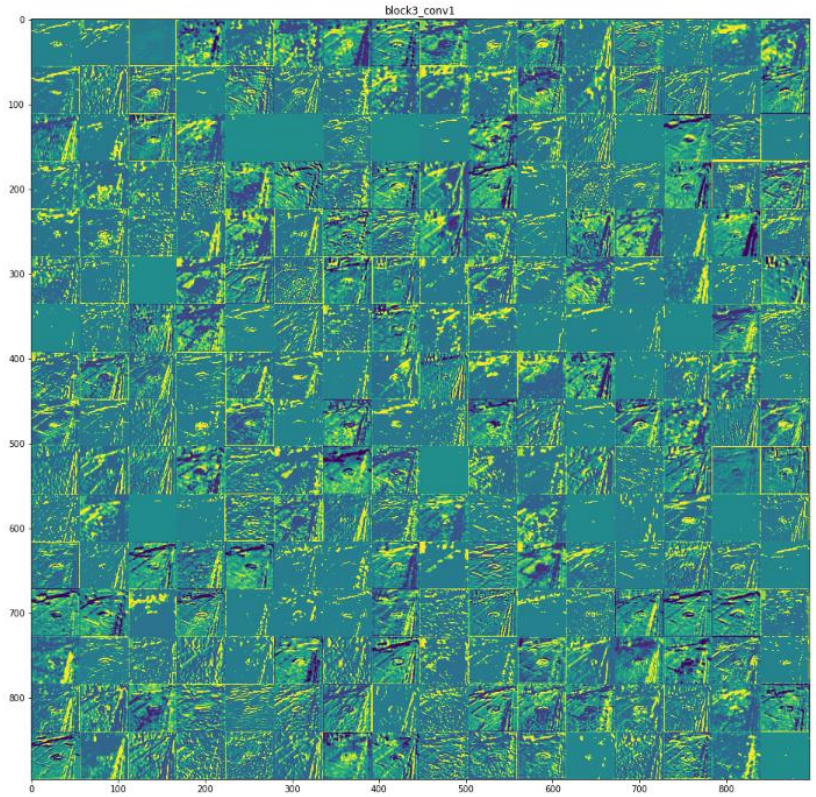
From block1, there are two convolutional layers and one max-pooling layer, we can clearly see the shape, the location and most information related to the pothole in the initial image. So in the first block, they acted like a collection of multiple edge detectors[11], keeping most of the features occurring in the original picture.

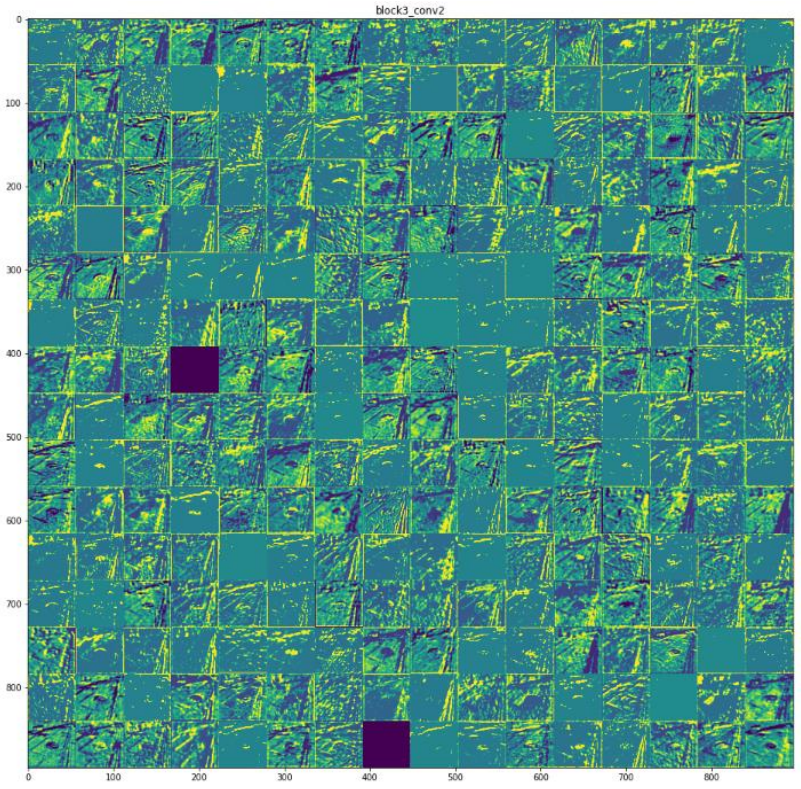

block2_conv1
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000
1200
1400
1600

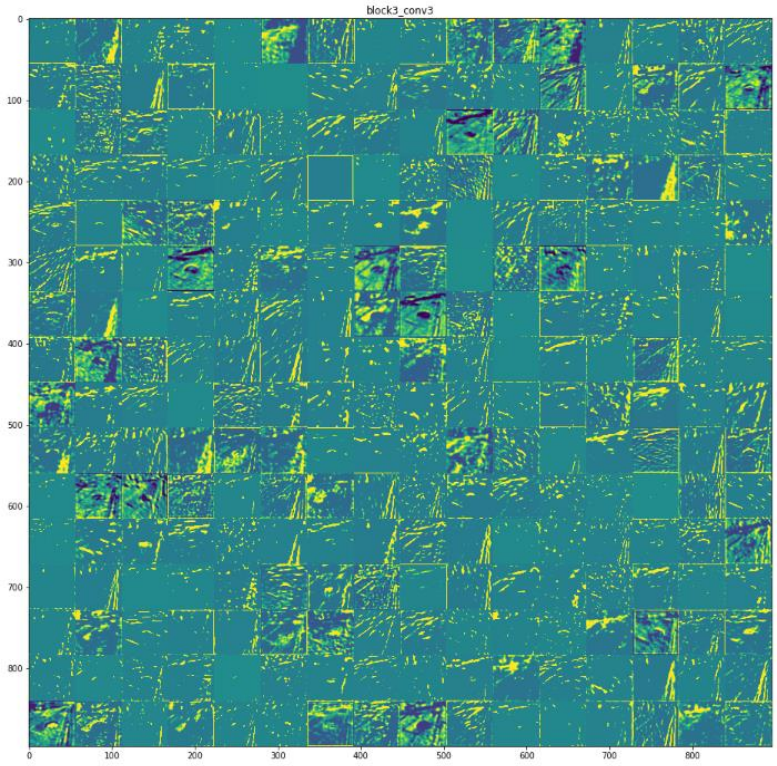

block2_conv2
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000
1200
1400
1600

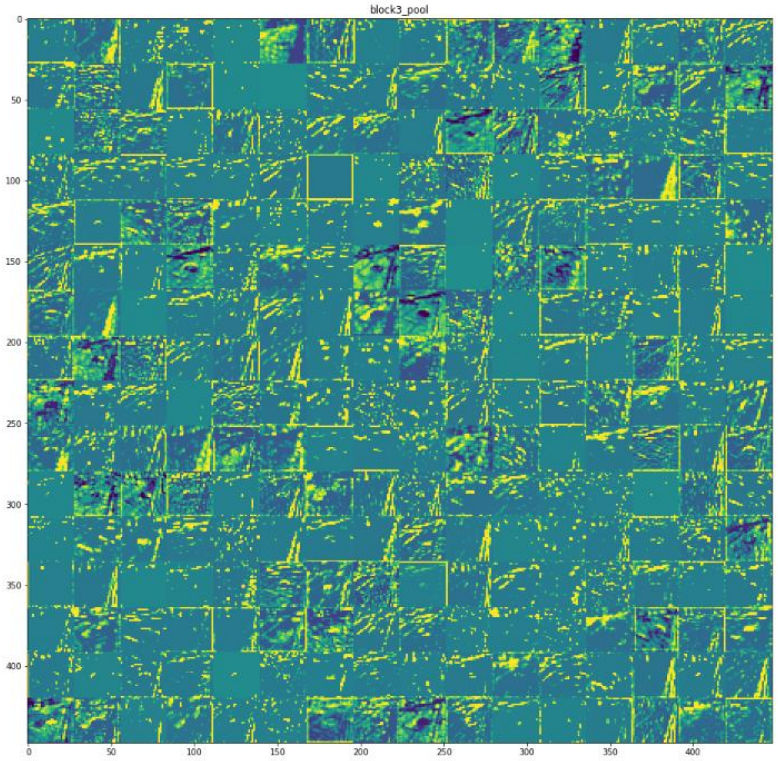


Block 2 also have two convolutional layers and one max-pooling layer, we can see that the channel increased to 128, which matches the 128 filters. Also there were some blanks in the block2_conv1, which means the sparse of the data and the filter didn't find the corresponding pattern in the image. The blank also showed up in deeper layers. From the feature maps, we can also see that the information related to the context of the pictures were not clear to be caught by the human eyes, but we still figured out the outline of the pothole.

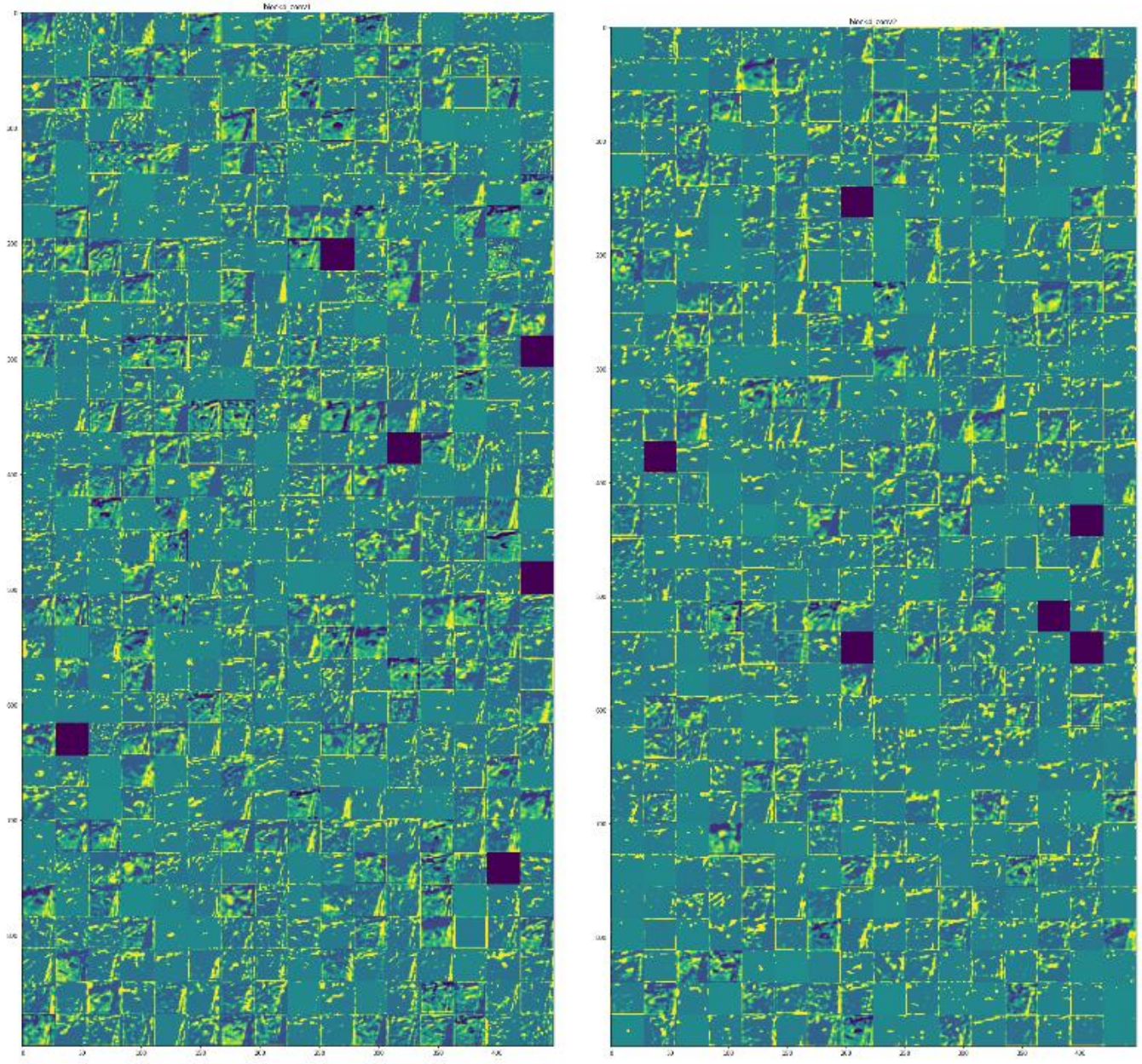

block3_conv1
0
100
200
300
400
500
600
700
800
0
100
200
300
400
500
600
700
800

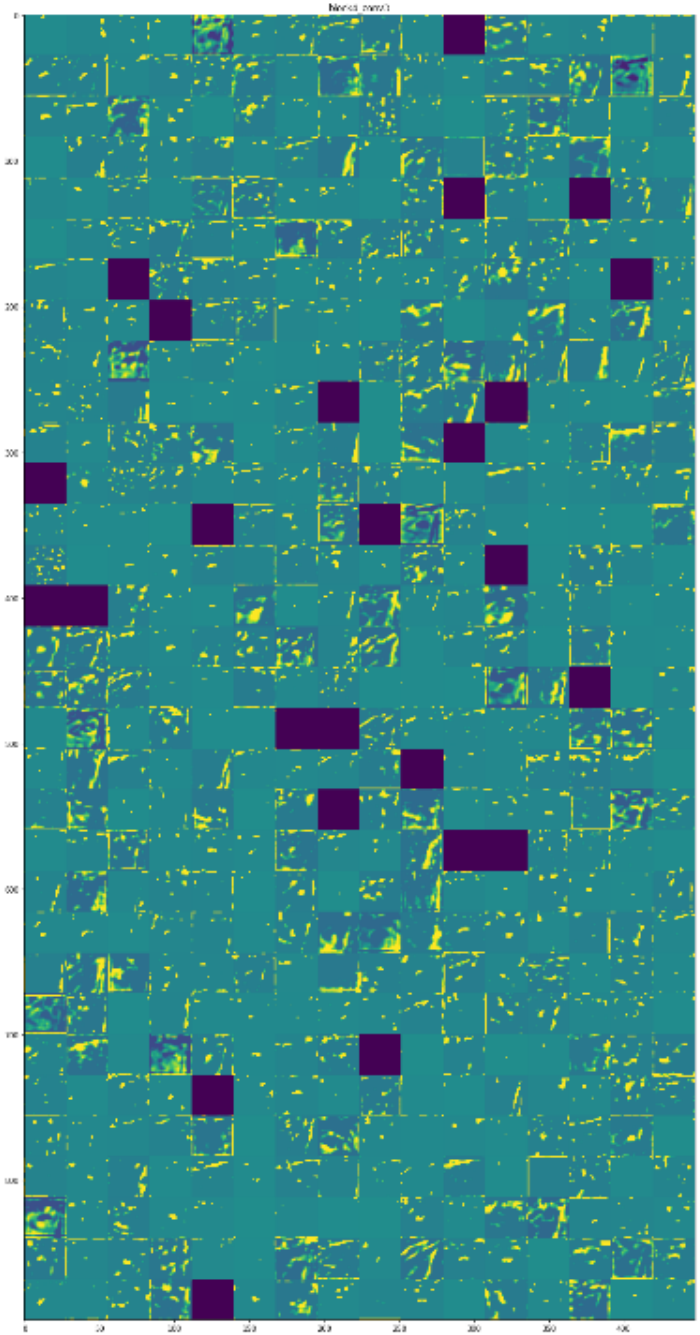

block3_conv2
0
100
200
300
400
500
600
700
800
0
100
200
300
400
500
600
700
800

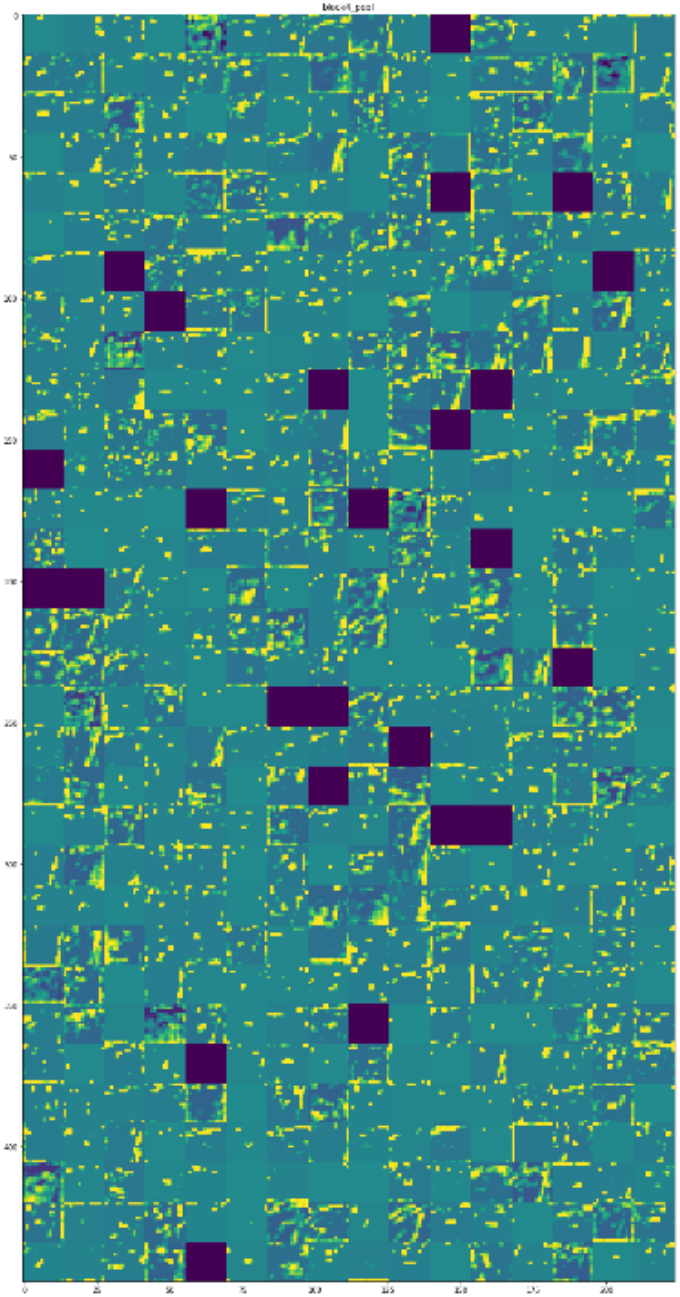


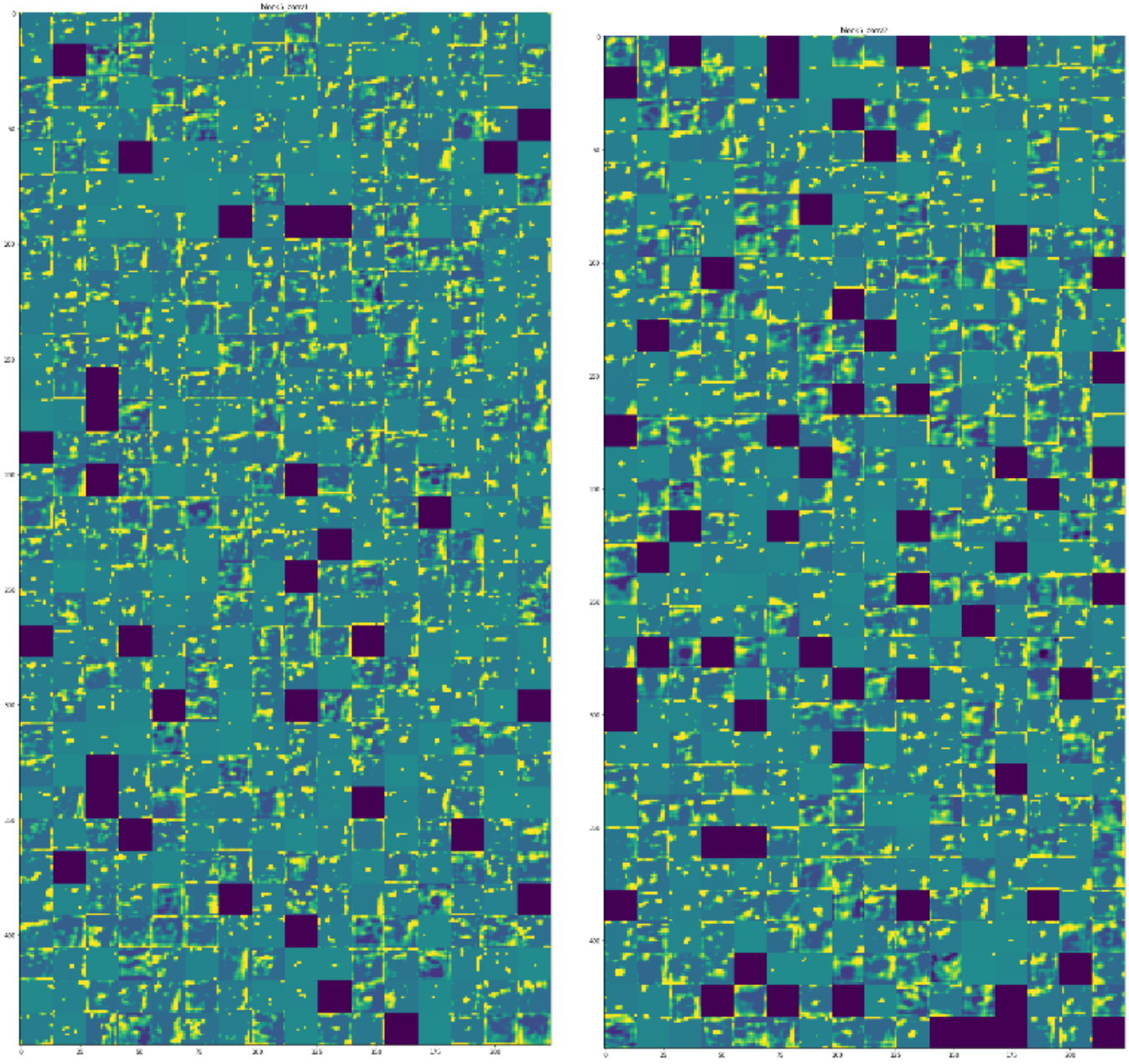


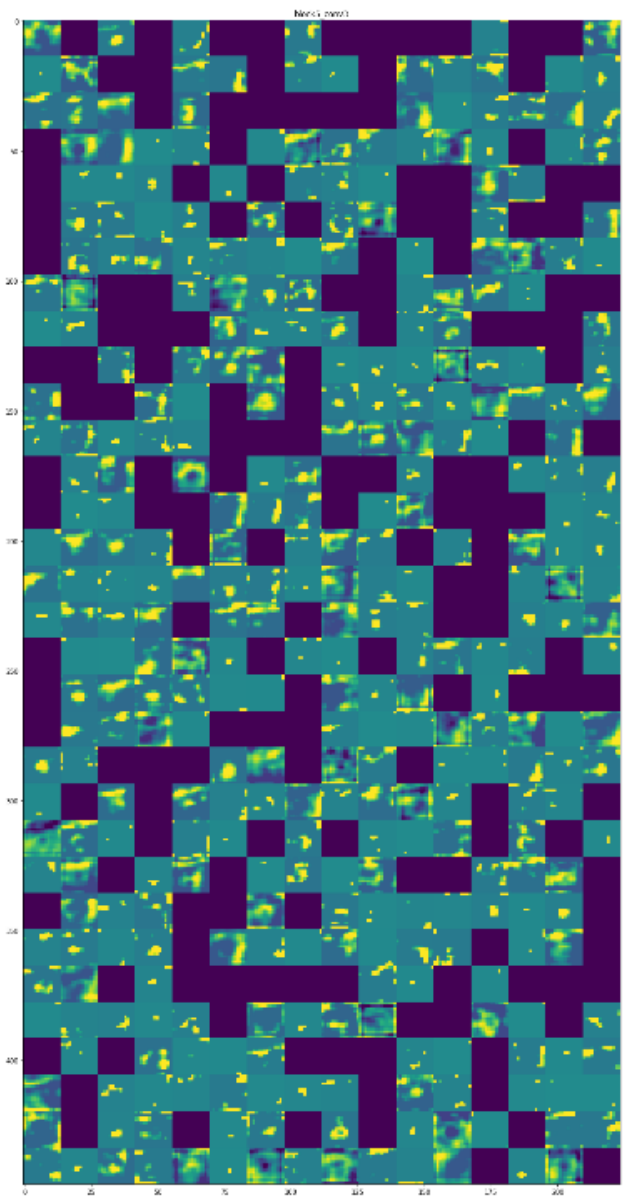
From block3, there are 3 convolutional layers and one max-pooling layer. The feature maps become more abstract and hard to interpret visually. So when the layers go deeper, the context of the image becomes trivial and very obscure. However, we can still figure out which is the pothole. At this stage, the model started to distinguish the different classes more clearly. It started to ignore the context and point out the key features belonging to our target class.

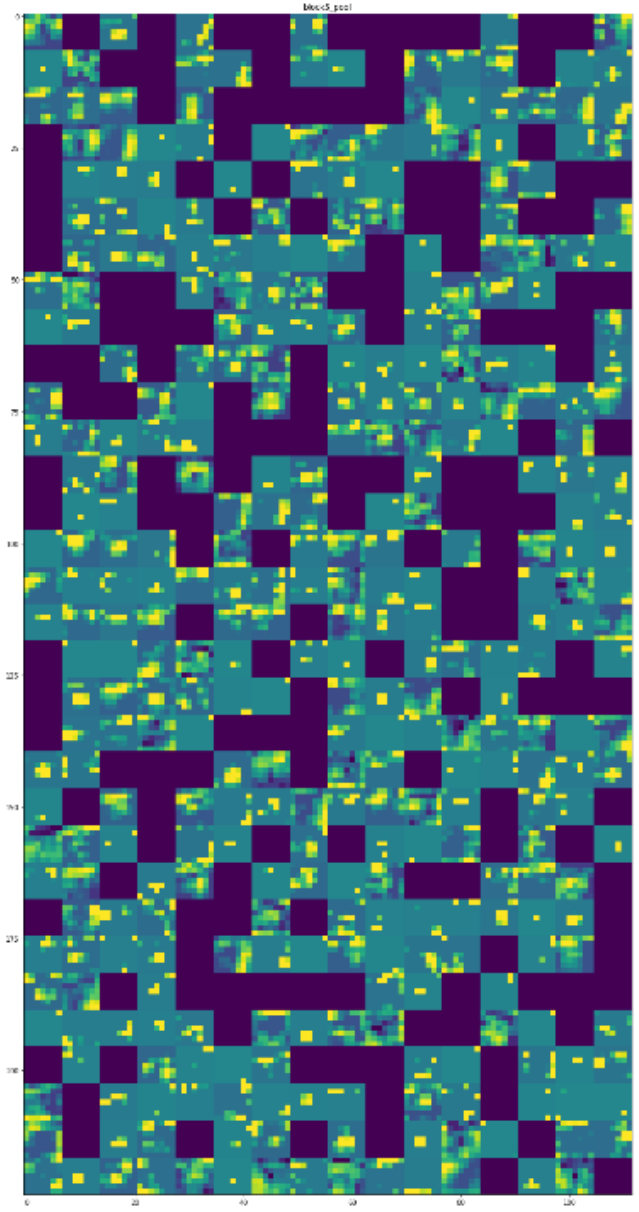












Block4 and block 5 also have 3 convolutional layers and one max_pooling layer. We can see that as the layers go deeper, more blanks showed up in the feature map and the filters increase to 512. This implies that our data becomes more sparse, it's normal the filter can not encode any matched pattern.

From the representation of the feature map layer by layer, we can know that the when the layer get deeper, the features extracted more abstract and not easily visualized. The model will care less about context and more about the target class. In other words, useless information is filtered out, and retain the important information, in order to predict the target class.

There are several technologies to interpret the CNN based model visually. In the future, we will strive to these technologies to make our CNN based model more interpretable.

Reference:

1.The Pothole Fact.Retrieved from https://www.pothole.info/the-facts/.

2. ALEX DAVIES.(2019,January 30).A Windshield-Mounted Cure for the Common Pothole. Retrieved from https://www.wired.com/story/carvi-pothole-detection/.

3. Image Preprocessing. Retrieved from https://keras.io/preprocessing/image/.

4.Understanding Support Vector Machine algorithm from examples (along with code).(2017, September 13). https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/.

5.A Short Introduction – Logistic Regression Algorithm.(2016, March 17). https://helloacm.com/a-short-introduction-logistic-regression-algorithm/.

6.Ilias Mansouri.Computer Vision Part 4: An overview of Image Classification architectures. https://medium.com/overture-ai/part-4-image-classification-9a8bc9310891.

7.VGG16 – Convolutional Network for Classification and Detection.(2018, November 20). https://neurohive.io/en/popular-networks/vgg16/.

8.Dipanjan (DJ) Sarkar.(2018, November 14).A Comprehensive Hands-on Guide to Transfer Learning with Real-World Applications in Deep Learning. https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a.

9.Django makes it easier to build better Web apps more quickly and with less code. Retrieved from https://www.djangoproject.com/.

10.Manish Prasad Thapliyal.Advance AI: Machine Learning design patterns.

https://medium.com/mlrecipies/advance-ai-machine-learning-design-patterns-d05397eb2191.

11.Himanshu Rawlani.(2018,December 27).Visual Interpretability for Convolutional Neural Networks.

https://towardsdatascience.com/visual-interpretability-for-convolutional-neural-networks-2453856210ce.