

AI Open Lab
MBC
(Malware Binary classification)
연구결과 보고서

임하늘(팀장)

이○○

김○○

<목차>

제 1장) 연구개발 수행 내용 및 결과

제1절 연구 개발 수행 내용

제2절 연구결과

1. Dataset 수집
2. feature extraction
3. model training or prediction

1절. 연구 개발 수행 내용

본연구는 Deep learning을 이용하여

Malware (Malicious + software)를 visualization하여 정적분석으로

Malware 를 classification 진행하는 연구로서

2가지 항목에 맞추어 진행 했다

1) Supervised learning

1. Convolutional Neural Network(CNN)

2) Unsupervised learning

1. Convolutional Autoencoder(CA)

2. Generative Adversarial Network(GAN)

연구 결론

각 file 마다 용량이 서로 틀려 feature extraction 생성할시 image size 가 제각각 틀려 image 를 규격화 해야하는 한계점을 발견 했다 image를 규격화 하는 과정에서 상대적으로 file이 큰 image feature 인 경우 다른 file 과 다르게 model 학습할 수 있는 feature 가 줄어들어 학습손실 되는 장애요점으로 오탐률을 피할 수 없었다

1). dataset 수집

KISA(한국인터넷진흥원)

R&D 데이터셋 목록

데이터셋 그룹	데이터셋	제공자	소개
악성코드 데이터셋	PC악성코드	한국인터넷진흥원 세인트시큐리티	악성코드 변종탐지, 그룹분류 기술, 성능평가에 활용된 5,045개의 악성코드 샘플과 분석결과
	지능형 악성코드	안랩, 하우리, 세인트시큐리티	2017 정보보호 R&D 데이터 챌린지 대회의 "악성코드 선제대응" 트랙에 활용된 300개의 지능형 악성코드
	대용량 정상, 악성파일 1 (2017 예선)	한국인터넷진흥원, 하우리, 세인트시큐리티	2017 정보보호 R&D 데이터 챌린지 대회의 "악성코드 탐지" 트랙 예선에 활용된 15000개의 대용량 정상, 악성파일
	대용량 정상, 악성파일 2 (2017 본선)	한국인터넷진흥원, 하우리, 세인트시큐리티	2017 정보보호 R&D 데이터 챌린지 대회의 "악성코드 탐지" 트랙 본선에 활용된 15000개의 대용량 정상, 악성파일
	대용량 정상, 악성파일 3 (2018)	한국인터넷진흥원, 안랩, 이스트시큐리티, 하우리, 세인트시큐리티	2018 정보보호 R&D 데이터 챌린지 대회의 "AI기반 악성코드 탐지"에 활용된 50000개 악성코드
	대용량 정상, 악성파일 4 (2019)	한국인터넷진흥원, 안랩, 이스트시큐리티, 하우리, 세인트시큐리티	2019 정보보호 R&D 데이터 챌린지 대회의 "AI기반 악성코드 탐지"에 활용된 40000개 악성코드
	VX Heaven 악성코드	호서대학교	VX heaven에서 배포하는, 15개의 악성코드 그룹으로 구성된 236,754개 악성코드
	메모리 상주 악성코드	한양대학교	실행파일 없이 메모리 상에 상주하는 악성코드 564개 및 분석 보고서



1st_answer



1st_answer.csv



KISA-CISC2017-Malware-1st.zip



대용량 악성코드 package 2개로 진행 했으며(각각 3GB)

악성코드 : 10500개 , 정상 : 4500개 총 15000개 진행

2). Feature extraction

.text section 추출 및 변환

PE 파일은 다양한 section 들로 이루어져 있다.

```
for files in file_list:
    current_path = os.path.join(path1, files)
    pe = pefile.PE(current_path)
    with open(current_path, 'rb') as f:
        for section in pe.sections:
            print('Name: ', section.Name)
            print('Location: ', section.PointerToRawData)
            print('Size: ', section.SizeOfRawData)
```

파이썬의 pefile 라이브러리를 통해 PE file속 section의 이름과 위치, 크기를 출력하였다. 그 결과는 다음과 같았다.

```
Name: b'.text\x00\x00\x00'
Location: 512
Size: 6656
Name: b'.rsrc\x00\x00\x00'
Location: 7168
Size: 1024
Name: b'.reloc\x00\x00'
Location: 8192
Size: 512
```

파일의 크기를 확인해보던 중 아래 사진과 같이 크기가 0인 섹션들이 존재함을 확인하였다.

```
Name: b'BSS\x00\x00\x00\x00\x00'
Location: 18432
Size: 0
```

따라서 text section 중 크기가 0인 PE file이 존재하는지 확인하는 과정을 진행하였다.

```
Name: b'.text\x00\x00\x00'
Location: 0
Size: 0
```

text section의 크기와 위치를 확인하는 코드를 추가하였다. 이미지 변환 작업을 하기 전 가급적이면 정사각형 이미지를 생성하고자 희망하였기 때문에 text section의 크기를 확인하였다.

```
for section in pe.sections:
    if b'.text' in section.Name:
        print(section.SizeOfRawData)
```

1024
16896
1039360

다양한 크기의 파일이 확인되었다. 가장 정확한 학습을 하기 위해 하드웨어가 처리할 수 있는 가장 큰 크기인 256*256에 맞추어 text section을 이미지로 변환하였다. 변환하는 코드는 다음과 같았다.

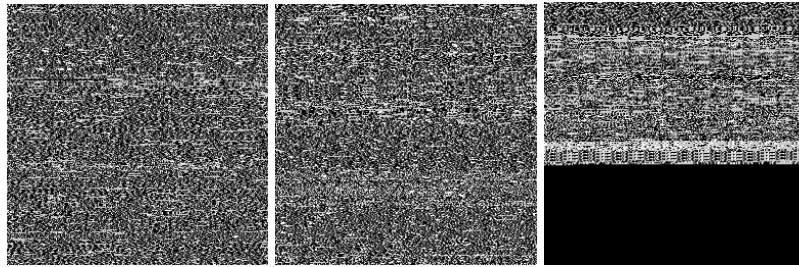
```
def transform(self):
    file_list = os.listdir(self.in_path)
    for files in file_list:
        current_path = os.path.join(self.in_path, files)
        pe = pefile.PE(current_path)
        with open(current_path, 'rb') as f:
            # .text 파일 찾기
            for section in pe.sections:
                if b'.text' in section.Name:
                    place = section.PointerToRawData
                    length = section.SizeOfRawData
                    if length > 0:
                        f.seek(place-1)
                        # .text 데이터 읽기
                        data = f.read(length)
                        if self.limit == True:
                            # 원하는 크기의 배열 만들기
                            ndarray = np.zeros(self.size*self.size)
                            # 배열 .text데이터로 채우기 % 정규화
                            for i in range(len(data)):
                                ndarray[i] = data[i]/255.0
                                if i == self.size*self.size-1:
                                    break
                            # 배열 차원 조절
                            ndarray = np.reshape(ndarray, (self.size, self.size))

            # 배열 저장
            np.save(os.path.join(self.out_path, files.split('.')[0]), ndarray)
```

위 코드는 다음과 같은 과정을 담고 있다.

- 1) data가 있는 디렉토리에 코드 작성
- 2) absolute directory 불러오기 (line 2 ~ line 3)
- 3) os에 적절하게 각 파일에 대한 경로 생성 (line 4)
- 4) 생성된 경로의 파일은 read byte 상태로 메모리에 로드 (line 5)
- 5) section 이름을 비교하며 .text가 존재하는지 확인 (line 8 ~ line 9)
- 6) .text section의 시작 위치와 크기 불러오기 (line 10 ~ line 11)
- 7) section의 크기가 이미지를 만들기에 적절한지 확인 (line 12)
- 8) .text section를 바이트 형태로 읽기 (line 13 ~ line 15)
- 9) 256*256만큼의 0으로 채워진 배열 생성 (line 18)
- 10) .text section 정규화 후 .npy로 저장 (line 20 ~ line 27)
- 11) 답지에 따라 0과 1로 카테고리에 따라 분류

코드를 통해 변환된 사진은 다음과 같았다.



위 사진에서 보이듯이 사진에 노이즈가 많아 단순하다는 것을 알 수 있다. 데이터의 품질을 확인하기 위해 CNN을 통해 학습률을 확인하였다. 그 결과 학습률이 매우 저조하였고 학습이 잘 진행되더라도 PE header 전체를 rescale한 데이터를 학습시켰을 때 보다 학습률이 낮았기에 PE header 전체를 이미지로 변환한 데이터를 사용하기로 하였다.

< 이미지 규격화 및 오차를 한계점>

1. 규격화에 자유로운 feature를 추출해서 학습을 돌려보려 했으나 파일마다 파일마다 PE Section의 존재여부, 사이즈 등이 다 다르며 크기가 다양하므로 feature를 뽑았을때 각각의 이미지 사이즈가 다를 수밖에 없다.
2. 이미지를 규격화 하지 않는이상 supervised learning의 한계점에 종착했다.
3. 실질적인 악성 분기가 많은 .text 섹션을 feature로 뽑는 방법과 전체 섹션을 feature로 뽑는 방법을 진행하였으며 둘다 동일하게 256x256 사이즈로 규격화 하여 진행하였다.

< 전체 Section을 256x256 사이즈의 이미지로 변환>

1. Python PEFile 모듈을 사용하며 PE 전체 header section의 데이터를 불러와 데이터 사이즈를 확인한다.

PE 전체 Section을 확인하면 다음과 같다.

<section 이름, section 주소, section RawData size>

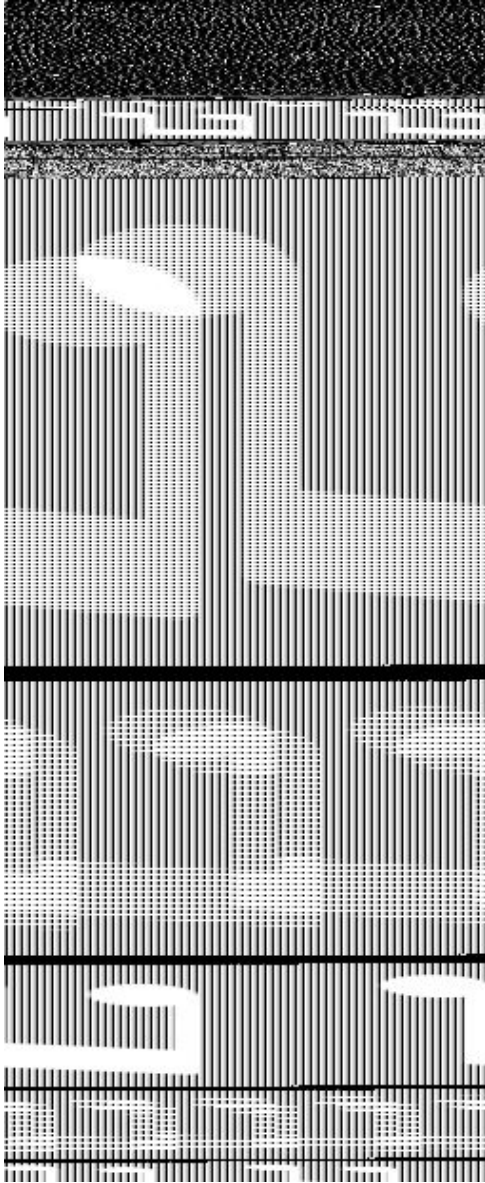
```
1 import pefile
2 pe = pefile.PE('8b4a9fbdaeba2dc8c48e40f28169db5e.vir')
3 for section in pe.sections:
4     print(section.Name, hex(section.VirtualAddress), section.SizeOfRawData)
5 #결과
6 '''
7 b'.text\x00\x00\x00' 0x480 2560
8 b'.rdata\x00\x00' 0xe80 256
9 b'.data\x00\x00\x00' 0xf80 1024
10 b'.edata\x00\x00' 0x1380 128
11 b'.INIT\x00\x00\x00\x00' 0x1400 256
12 b'.reloc\x00\x00' 0x1500 256
13 '''
```

파일마다 section이 다르며 어떤 파일에는 존재하지 않거나 섹션의 RawData 사이즈가 0인 경우 등 다양한 경우가 존재한다.

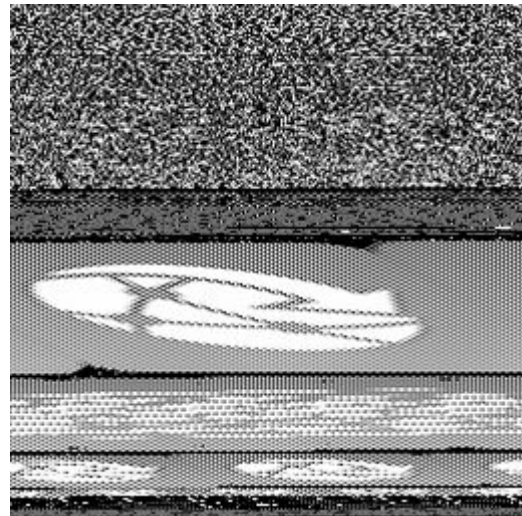
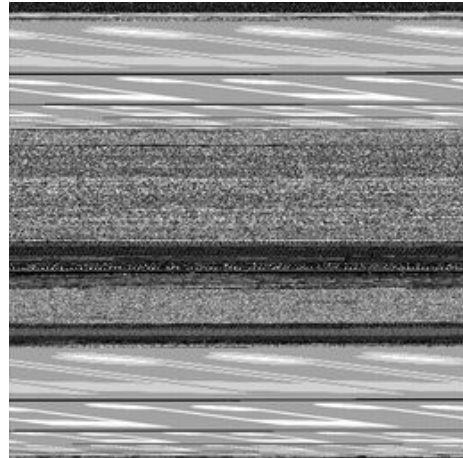
4. 한번의 길이를 루트로 계산한다.

5. PIL 라이브러리를 이용하여 위에서 구한 길이로 정사각형의
사이즈로 image를 만든다.
(대부분 256x256 보다 큰경우가 대부분이지만 작은경우도 존재)

<사이즈 규격화 하기 전>

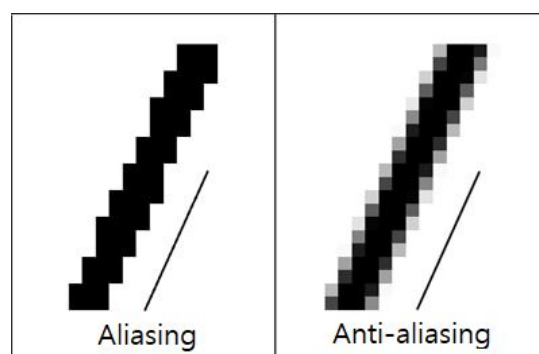


<규격화 후>



6. 만든 이미지를 256x256 사이즈로 변환하기 위해 PIL에 anti aliasing
filter 를 이용하여 이미지 사이즈를 규격화 시킨다.

< Anti-aliasing 예시 >



< malware 파일로부터 Image 및 numpy 변환 코드 >

```
1 import pefile
2 import os
3 import math
4 import numpy as np
5 from PIL import Image
6
7 class exe_visual:
8
9     # PE 헤더 데이터로부터 이미지, numpy 형식 파일 저장
10     def visualization_malware(self, dataset, outputfile, imagesize, width=0):
11         global size
12
13         # 헤더 데이터로부터 정사각형 한변의 길이 구하기
14         if (width == 0):
15             size = len(dataset)
16             width = math.ceil( math.sqrt(size) )
17             heigher = int(size / width) + 1
18
19         #이미지 생성 및 데이터 입력
20         image = Image.new("L", (width, heigher))
21         image.putdata(dataset)
22
23         #이미지 사이즈 조절(ANTIALIAS 사용) 및 이미지 저장
24         image = image.resize((imagesize, imagesize), Image.ANTIALIAS)
25         imagename = outputfile + ".png"
26         image.save(imagename)
27
28         #numpy array 형식 형변환 및 0~1 normalization 처리 저장
29         nparray = np.asarray(image)
30         nparray = nparray/imagesize
31         np.save(outputfile, nparray)
32
33         #image.show()
34
35     #inputPath 에 있는 모든 exe, vir확장자 파일을 읽어
36     #visualization_malware 로 데이터 넘김
37     def make_png(self, inputpath, imagesize=256, section=""):
38         files = []
39         if os.path.isfile(inputpath):
40             files.append(inputpath)
41         elif os.path.isdir(inputpath):
42             for file in os.listdir(inputpath):
43                 if file.endswith(".exe") or file.endswith(".vir"):
44                     files.append(inputpath + "/" + file)
45         else:
46             raise Exception(inputpath + " is not File or directory")
47
48         for file in files:
49             try:
50                 pe = pefile.PE(file)
51                 data = bytearray()
52                 for section in pe.sections:
53                     name = section.Name
54                     #지정된 섹션이 있으면 해당 섹션만 가져오며
55                     #없으면 전체 데이터 저장
56                     if not section:
57                         #섹션 이름에 있는 앞뒤 \x00 문자 제거
58                         if section.Name.decode().rstrip('\x00') == section:
59                             data = data + section.get_data()
60                     else:
61                         data = data + section.get_data()
62                 self.visualization_malware(dataset=data, imagesize=imagesize, outputfile=file)
63             except Exception as e:
64                 print('Exception catch ' + file, e)
```


Numpy 로 데이터셋 만들시 255로 나누어 0~1 사이의 값으로 정규화 처리를 하였다.















7. 최종 만들어진 파일을 answer.csv 파일을 읽어 정상 파일과 malware 파일을 분리

< 정상파일과 malware 파일 분리 코드 >

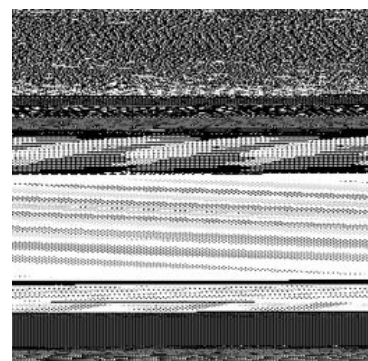
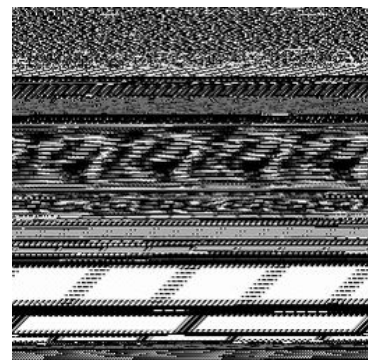
```
1 import os
2 import csv
3
4 #answer.csv 파일을 열어 악성파일과 아닌 파일 0,1 폴더로 나눠 분리
5 with open('1st_answer.csv') as csvfile:
6     reader = csv.reader(csvfile)
7     for line in reader:
8         os.rename(line[0] + ".npz", line[1] + "/" + line[0] + ".npz")
9
10 with open('2nd_answer.csv') as csvfile:
11     reader = csv.reader(csvfile)
12     for line in reader:
13         os.rename(line[0] + ".npz", line[1] + "/" + line[0] + ".npz")
```

8. 총 일반파일 4,500 개 / malware 10,500 개 데이터셋 생성

< 만들어진 데이터셋 >

 KISA_numpy.7z.001	2 days ago
 KISA_numpy.7z.002	2 days ago
 KISA_numpy.7z.003	2 days ago
 KISA_numpy.7z.004	2 days ago
 KISA_numpy.7z.005	2 days ago
 KISA_numpy.7z.006	2 days ago
 KISA_numpy.7z.007	2 days ago
 KISA_numpy.7z.008	2 days ago
 KISA_numpy.7z.009	2 days ago
 KISA_numpy.7z.010	2 days ago
 KISA_numpy.7z.011	2 days ago
 KISA_numpy.7z.012	2 days ago
 KISA_numpy.7z.013	2 days ago
 KISA_numpy.7z.014	2 days ago

<규격화된 Image예시>



3.) Model training 결과

Supervised learning

1. Convolutional Neural Network(CNN)

Image classification 에 가장 대중적으로 알려져있는 CNN 를
training 이후 prediction 결과

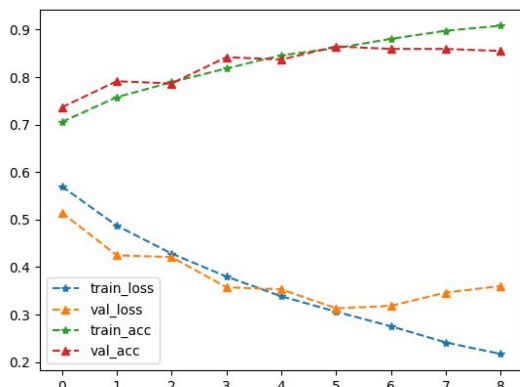
```
data_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'data', '0')
for files in tqdm(os.listdir(data_dir)):
    p = os.path.join(data_dir, files)
    n = np.load(p)
    data.append([n, '0'])
    number+=1

data_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'data', '1')
for files in tqdm(os.listdir(data_dir)):
    p = os.path.join(data_dir, files)
    n = np.load(p)
    data.append([n, '1'])
    number+=1

random.shuffle(data)
X_train, y_train, X_test, y_test = [], [], [], []
for i in range(0, int(number*0.8)):
    X_train.append(data[i][0])
    y_train.append(data[i][1])
for i in range(int(number*0.8), number):
    X_test.append(data[i][0])
    y_test.append(data[i][1])
del(data)
X_train = np.array(X_train)
y_train = np.array(y_train)
X_test = np.array(X_test)
y_test = np.array(y_test)
```

```
X_train = X_train.astype("float32")
X_test = X_test.astype("float32")
X_train = np.reshape(X_train, (len(X_train), 256, 256, 1))
X_test = np.reshape(X_test, (len(X_test), 256, 256, 1))
```

```
y_train = to_categorical(y_train, number_classes)
y_test = to_categorical(y_test, number_classes)
```



5번째 이후 과적합 발생으로

학습 조기종료 발생

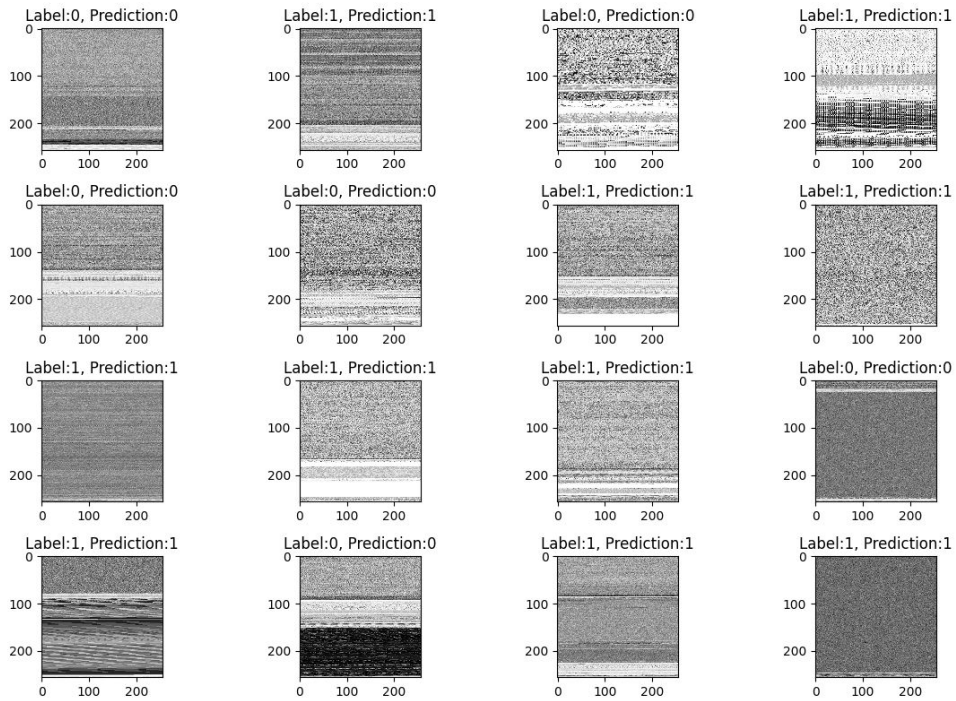
```
def MalwareModelCnn():
```

```
    model = Sequential()
    model.add(Conv2D(64, kernel_size=(3, 3), padding="same", activation='relu'))
    model.add(MaxPooling2D((2, 2), padding="same"))
    model.add(Conv2D(64, kernel_size=(3, 3), padding="same", activation='relu'))
    model.add(MaxPooling2D((2, 2), padding="same"))
    model.add(Conv2D(32, kernel_size=(3, 3), padding="same", activation='relu'))
    model.add(MaxPooling2D((2, 2), padding="same"))
    model.add(Conv2D(16, kernel_size=(3, 3), padding="same", activation='relu'))
    model.add(Flatten())
    model.add(Dense(128, activation="relu"))
    model.add(Dropout(0.3))
    model.add(Dense(32, activation="relu"))
    model.add(Dense(10, activation="relu"))
    model.add(Dense(2, activation="sigmoid"))
```

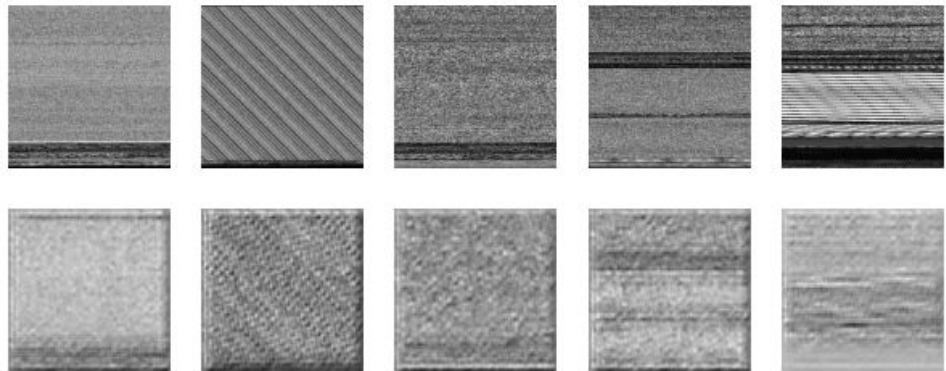
```
    model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["acc"])
    return model
```

```
result > 10180
loss > 1126
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 256, 256, 64)	640
max_pooling2d (MaxPooling2D)	(None, 128, 128, 64)	0
conv2d_1 (Conv2D)	(None, 128, 128, 32)	18464
max_pooling2d_1 (MaxPooling2D)	(None, 64, 64, 32)	0
conv2d_2 (Conv2D)	(None, 64, 64, 16)	4624
max_pooling2d_2 (MaxPooling2D)	(None, 32, 32, 16)	0
conv2d_3 (Conv2D)	(None, 32, 32, 8)	1160
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 64)	524352
dropout (Dropout)	(None, 64)	0
dense_1 (Dense)	(None, 32)	2080
dense_2 (Dense)	(None, 10)	330
dense_3 (Dense)	(None, 2)	22
Total params: 551,672		
Trainable params: 551,672		
Non-trainable params: 0		



2. Convolutional autoencoder



ACC 재현율이 37% ~ 40% 이상으로 재현율을 올리기 힘들었으면
그 이유는 IMAGE 자체가 단순하여 Feature 추출에 한계점이 있는걸로
추측한다

3. Generative Adversarial Network(GAN)

model 생성망 기준으로 하여 fake image 를 계속 생성하는 생성망 모델을 만들고 부족한 data 를 추가하여 진행을 해보았다

Layer (type)	Output Shape	Param #
dense_4 (Dense)	(None, 1024)	67109888
leaky_re_lu_3 (LeakyReLU)	(None, 1024)	0
dropout (Dropout)	(None, 1024)	0
dense_5 (Dense)	(None, 512)	524800
leaky_re_lu_4 (LeakyReLU)	(None, 512)	0
dropout_1 (Dropout)	(None, 512)	0
dense_6 (Dense)	(None, 256)	131328
leaky_re_lu_5 (LeakyReLU)	(None, 256)	0
dropout_2 (Dropout)	(None, 256)	0
dense_7 (Dense)	(None, 1)	257
Total params: 67,766,273		
Trainable params: 67,766,273		
Non-trainable params: 0		

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	25856
leaky_re_lu (LeakyReLU)	(None, 256)	0
dense_1 (Dense)	(None, 512)	131584
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0
dense_2 (Dense)	(None, 1024)	525312
leaky_re_lu_2 (LeakyReLU)	(None, 1024)	0
dense_3 (Dense)	(None, 65536)	67174400
Total params: 67,857,152		
Trainable params: 67,857,152		
Non-trainable params: 0		

생성망 모델을 만듬에 있어 Cluster 목적으로 진행하였다

(각각 epochs 100 단위로 총 400번을 학습한 결과)

image 자체가 단순하여 model를 생성하여 부족한 데이터를 충족해준다 하더라도 오탐률은 피할 수 없다는걸 확인

