EXE FILE TO IMAGE

CNNs offer a potent approach to malware identification, leveraging deep learning
techniques to automatically learn and detect patterns indicative of malicious software. By understanding the architecture, training process, and deployment considerations of CNN-based malware identification systems, CNN is effective with
detecting source code and binary code, it can further identify malware that is embedded into benign code, leaving malware no place to hide. The works that use
CNNs, typically converts malware binaries to digital images and pass them into a
CNN in order to detect malware. the raw bytes of an EXE file to create an image of
the EXE. The byte stream

E4	C0	56
A3	D2	78
56	A3	FF

n an array then transform
it into an image like in [Le yte sequence: E4 C0 56 A3
D2 78 56 A3 FF, can be re x:

Gray scale image for the a sequence of bytes

The previous matrix will be transformed into a Gray image where each box of the
matrix will be a pixel in the image (degrees of Gray between 0 and 255)

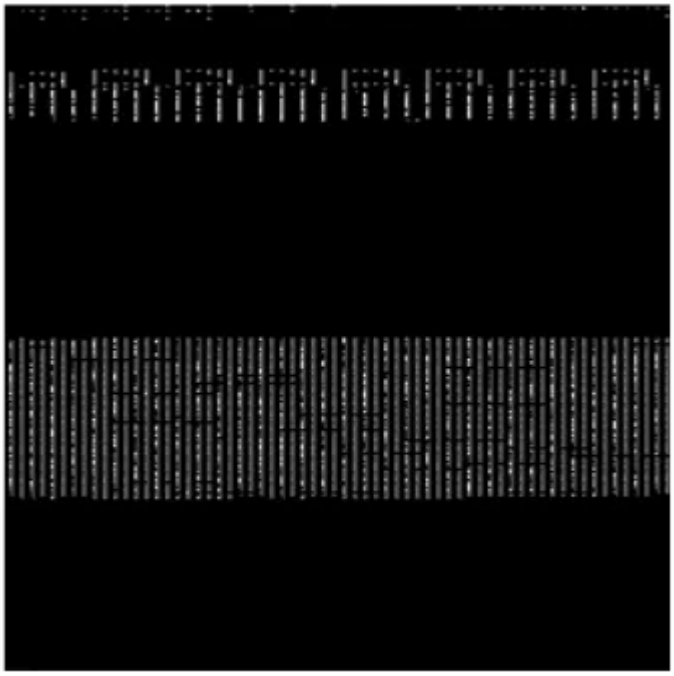


Image of a benign file 128*128

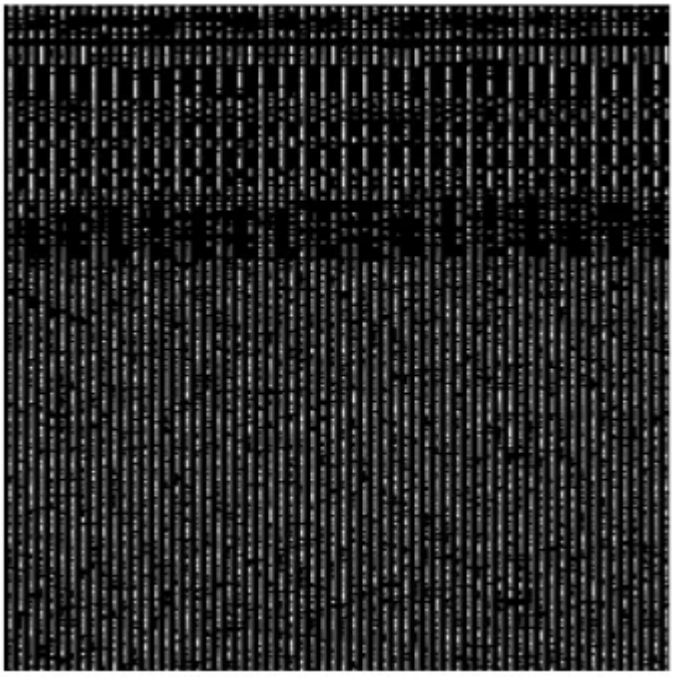


Image of a malware 128*128

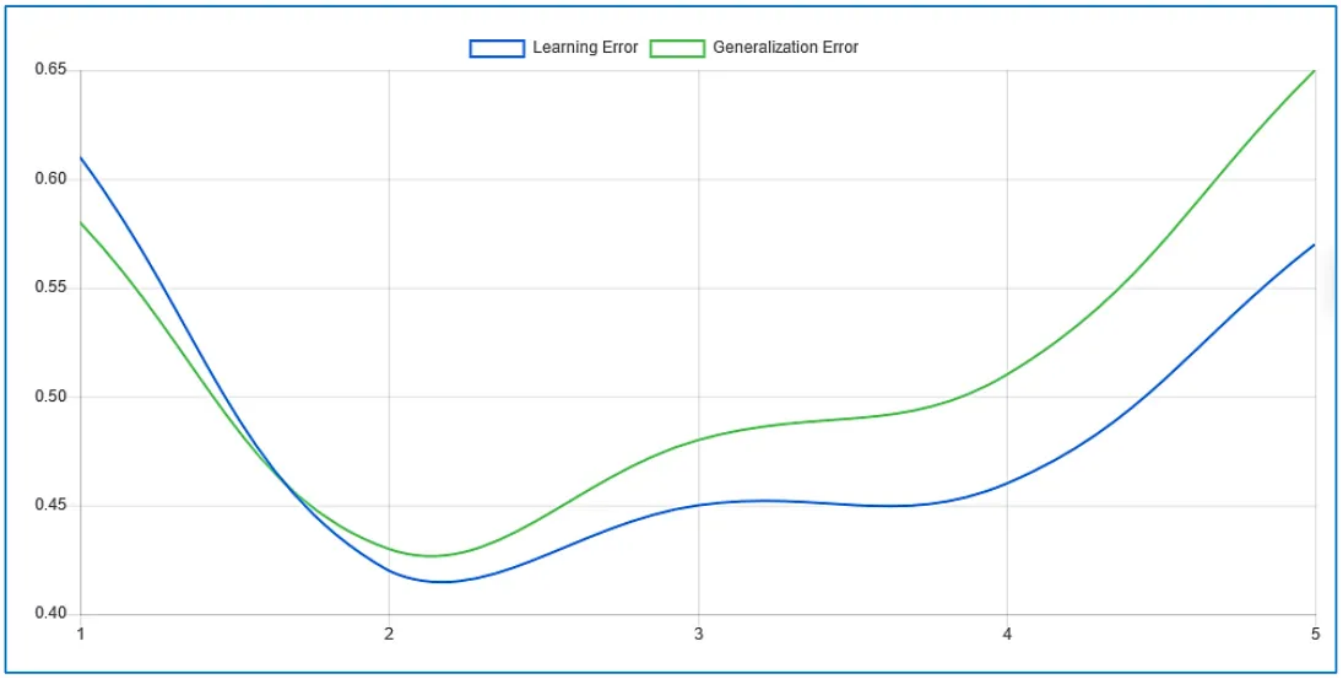
Let’s do some coding and implement this first part.

```
from os import listdir
from PIL import Image
import os.path
import numpy as np

path = 'path/to/folder/containing/exe/files'

h = 256 #height of image
w = 256 #width of image

#be careful with using this function, it will consume memory, access to disk and time
images = []
for f in listdir(path_new):
  with open(os.path.join(path_new, f), 'rb') as img_set:
    img_arr = img_set.read(h*w)
    while img_arr:
      if len(img_arr) == h*w and img_arr not in images:
        images.append(img_arr)
      img_arr = img_set.read(h*w)

#And you can save them into png files
count = 0
for img in images:
  png = Image.fromarray(np.reshape(list(img), (h,w)).astype('float32'), mode='L')
  png.save('image_l%d.png' %count)
  count += 1
```

And this is how you transform an EXE file into a set of images of size h*w

CREATING CNN

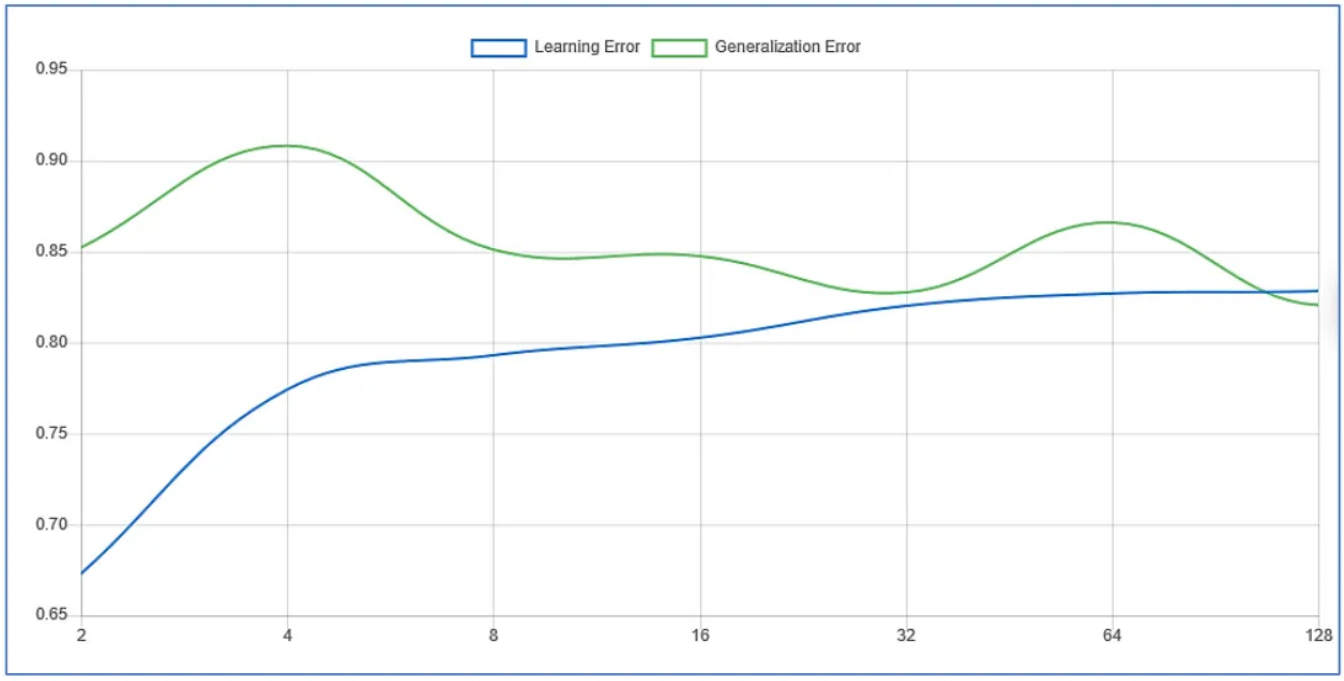
Now we come to the point of designing a convolution neural network to analyze the images and predict the maliciousness of a file. When it comes to the design of a CNN there are a lot of parameters that could be taken into consideration: Number of layers image size, kernel size, number of kernels, number of layers… We did a little experiment where we varied all parameters including number of layers and we projected the average for each number of layers. The following plot gives the minimal learning error achieved by each neural network of a specific number of layers ranging from 1 to 5.



Variation of learning error and generalization error for each number of hidden layers

Number of filters

We did similar experience as the previous one, results are presented in the plot
below. We note that we did not vary the size of a filter, we took the usual used sizes

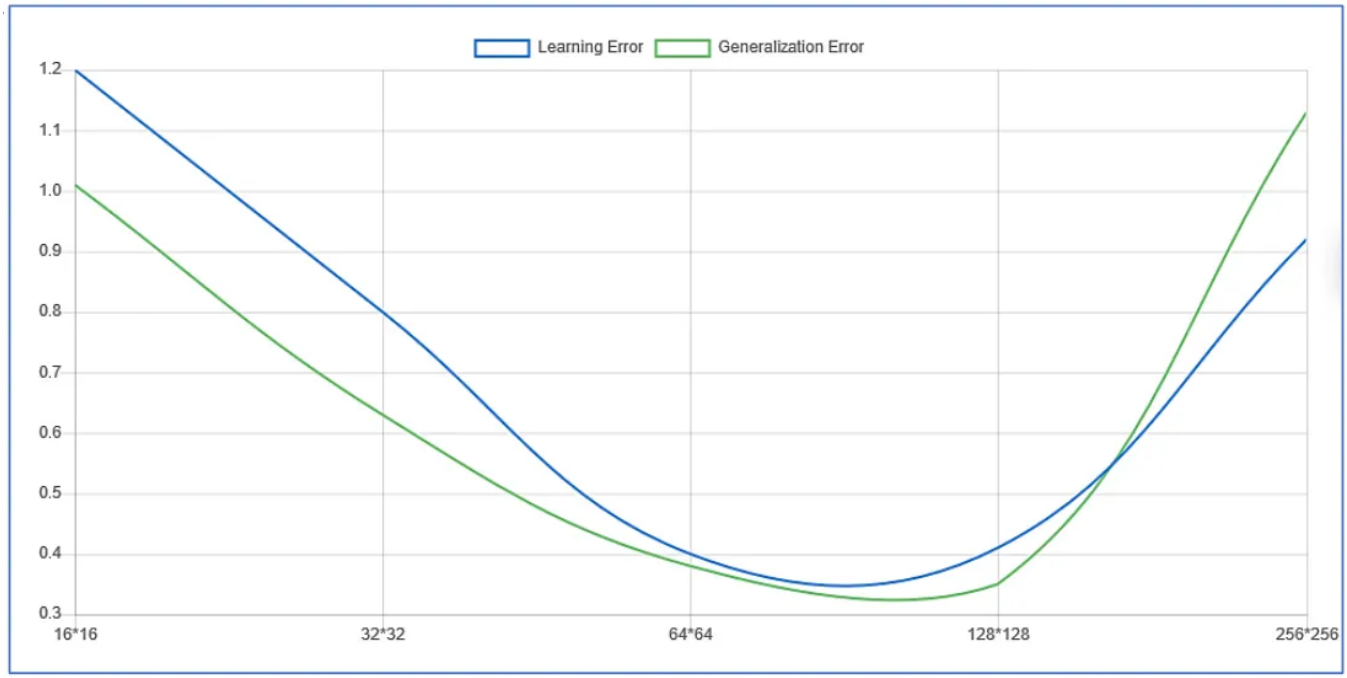


Variation of learning error and generalization error for each number of filters

Size of image

Size of the image is another parameter, while it’s not obvious what will work best in
practice we can already eliminate some choices. Choosing a big size will make it
difficult for learning since the input layer of the CNN will be so big and thus

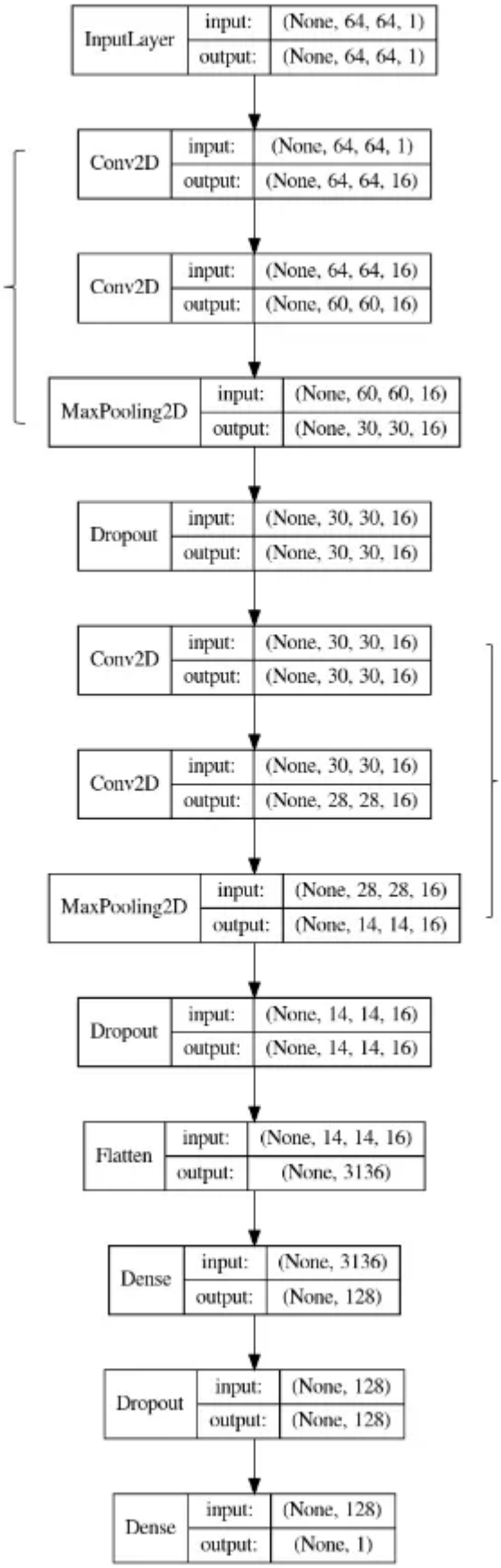
need more hidden layers which will result in a shortage of memory or long calculations. Also choosing large dimensions will induce the problem of small EXE
which doesn’t fill even one image. In the other hand choosing small dimensions will
cover most of EXE files but it will break some localities inside the EXE file since we



Errors by size of image (lazy title right?)

Architecture of CNN

From previous sections we combine best parameters to design our CNN: 2 Hidden
layers, 16 filters and 64*64 images.
The architecture of the network will look like that. Note that a convolution layer can
be composed of multiple conv2d layers followed by pooling layer and maybe drop
out. So in this example we have multiple conv2d layers grouped in two convolution
layers. The flatten layers is used to transform the output into one long vector then
process it by a classification layer.



First of all, we do some reshaping and we transform the image into gray scale and
also import essential modules.

```
import keras
from keras.models import Sequential
from keras.layers import Dense , Dropout , Activation , Flatten
from keras.layers import Conv2D , MaxPooling2D
import numpy as np

#reshape images to fit into the CNN model
img_list = np. zeros (shape = (len (images), h,w, 1), dtype = np.uint8)
for j in range (len (images)):
    img_list[j,:,:, 0] = np. reshape (list (images[j]), (h,w))

img_list = img_list. astype ('float32' )
img_list /= 255
```

After that we implement the architecture:

```
model = Sequential ()
#Conv2D Layers
model. add (Conv2D (12 , (25 , 25 ), padding = 'same' ,input_shape = img_list.shape[ 1:], activatio
model. add (Conv2D (12 , (25 , 25 ), activation = 'relu' ))
#Max Pooling Layer
model. add (MaxPooling2D (pool_size = (2, 2)))
#Conv2D Layer
model. add (Conv2D (12 , (13 , 13 ), padding = 'same' , activation = 'relu' ))
model. add (Conv2D (12 , (13 , 13 ), activation = 'relu' ))
#Max Pooling
model. add (MaxPooling2D (pool_size = (2, 2)))
#Flattening Layer
model. add (Flatten ())
#Dense Layer
model. add (Dense (1024 , activation = 'relu' ))
model. add (Dense (1, activation = 'sigmoid' ))

model. compile (loss = 'binary_crossentropy' ,
            optimizer = 'adam' ,
            metrics = ['binary_accuracy' ])

model. summary ()
```

code for cnn