CNN implementation in Naani

June 9, 2024

Contents

Chapter 1

CNN in Naani

CNN is a type of deep learning model for processing data that has a grid pattern, such as images and designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns. CNN is a mathematical construct that is typically composed of three types of layers (or building blocks): convolution, pooling, and fully connected layers. The first two, convolution and pooling layers, perform feature extraction, whereas the third, a fully connected layer, maps the extracted features into final output, such as classification.

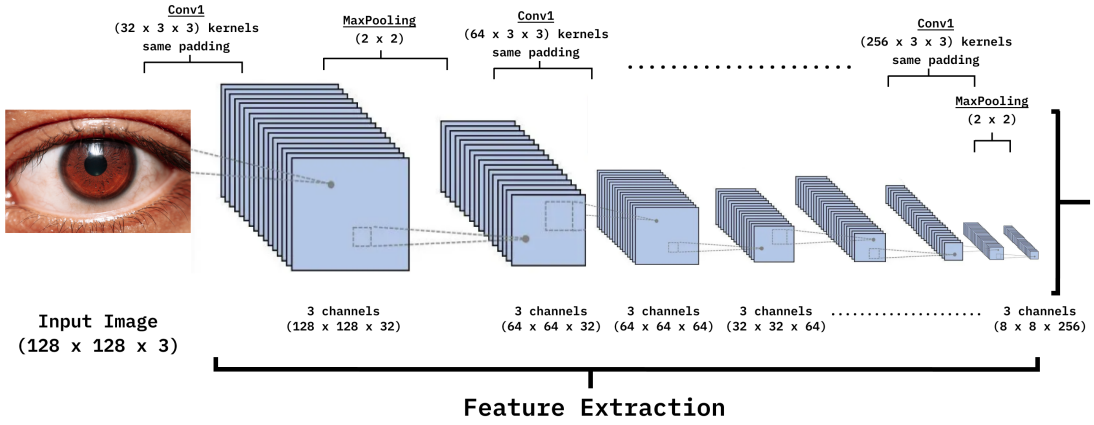

Figure 1.1: CNN architecture in Naani I

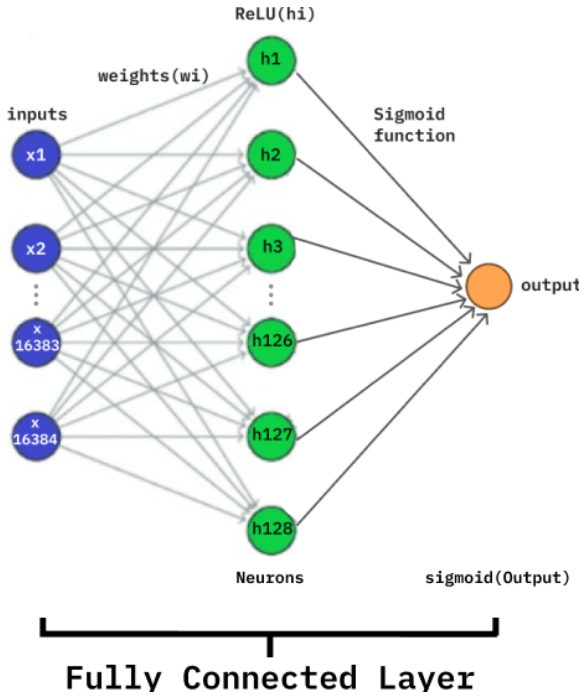

Figure 1.2: CNN architecture in Naani II

1.1 Convolutional Layer

1.1.1 Image Resizing and kernel Initialization

Memory Calculation for Images at Different Resolutions

64x64 Image:

$$\begin{aligned}\text{Memory per image} &= 64 \times 64 \times 3 \times 4 \text{ bytes} \\ &= 49,152 \text{ bytes} \\ &\approx 48 \text{ KB} \\ \text{Memory for 8000 images} &= 8000 \times 48 \text{ KB} \\ &= 384,000 \text{ KB} \\ &= 384 \text{ MB}\end{aligned}$$

128x128 Image:

$$\begin{aligned}\text{Memory per image} &= 128 \times 128 \times 3 \times 4 \text{ bytes} \\ &= 196,608 \text{ bytes} \\ &\approx 192 \text{ KB} \\ \text{Memory for 8000 images} &= 8000 \times 192 \text{ KB} \\ &= 1,536,000 \text{ KB} \\ &= 1,536 \text{ MB} \\ &= 1.5 \text{ GB}\end{aligned}$$

256x256 Image:

$$\begin{aligned}
\text{Memory per image} &= 256 \times 256 \times 3 \times 4 \text{ bytes} \\
&= 786,432 \text{ bytes} \\
&\approx 768 \text{ KB} \\
\text{Memory for 8000 images} &= 8000 \times 768 \text{ KB} \\
&= 6,144,000 \text{ KB} \\
&= 6,144 \text{ MB} \\
&= 6 \text{ GB}
\end{aligned}$$

Input images are resized to 128x128 pixels with 3 color channels (Red, Green, and Blue, RGB). This resizing involves creating three separate 128x128 matrices for each color channel. Each element within these matrices represents the intensity of the respective color channel for a specific pixel in the image. The final output image is constructed by stacking these three matrices along the third dimension, resulting in a 128x128 image with 3 color channels.

$$resizedImage = \begin{bmatrix} (r_{11}, g_{11}, b_{11}) & (r_{12}, g_{12}, b_{12}) & \cdots & (r_{1,128}, g_{1,128}, b_{1,128}) \\ (r_{21}, g_{21}, b_{21}) & (r_{22}, g_{22}, b_{22}) & \cdots & (r_{2,128}, g_{2,128}, b_{2,128}) \\ \vdots & \vdots & \ddots & \vdots \\ (r_{128,1}, g_{128,1}, b_{128,1}) & (r_{128,2}, g_{128,2}, b_{128,2}) & \cdots & (r_{128,128}, g_{128,128}, b_{128,128}) \end{bmatrix}$$

For example: The fig 1.1 is resized into a 128×128 pixel of 3 channels.
Red Channel Matrix:

$$\begin{bmatrix} 166 & 166 & 168 & \dots & 214 & 217 & 214 \\ 167 & 171 & 173 & \dots & 215 & 216 & 213 \\ 164 & 170 & 178 & \dots & 219 & 214 & 206 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 200 & 197 & 197 & \dots & 214 & 212 & 209 \\ 199 & 196 & 195 & \dots & 214 & 212 & 208 \\ 200 & 196 & 194 & \dots & 211 & 212 & 212 \end{bmatrix}$$

Green Channel Matrix:

$$\begin{bmatrix} 91 & 89 & 92 & \dots & 152 & 157 & 153 \\ 85 & 89 & 99 & \dots & 153 & 157 & 151 \\ 79 & 89 & 110 & \dots & 161 & 155 & 145 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 116 & 111 & 109 & \dots & 156 & 151 & 146 \\ 114 & 108 & 105 & \dots & 155 & 152 & 145 \\ 122 & 112 & 106 & \dots & 150 & 151 & 148 \end{bmatrix}$$

Blue Channel Matrix:

$$\begin{bmatrix} 67 & 65 & 69 & \dots & 118 & 123 & 120 \\ 59 & 63 & 74 & \dots & 119 & 124 & 117 \\ 54 & 63 & 86 & \dots & 130 & 123 & 111 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 84 & 79 & 77 & \dots & 124 & 117 & 110 \\ 83 & 77 & 73 & \dots & 123 & 118 & 109 \\ 91 & 81 & 76 & \dots & 115 & 115 & 112 \end{bmatrix}$$

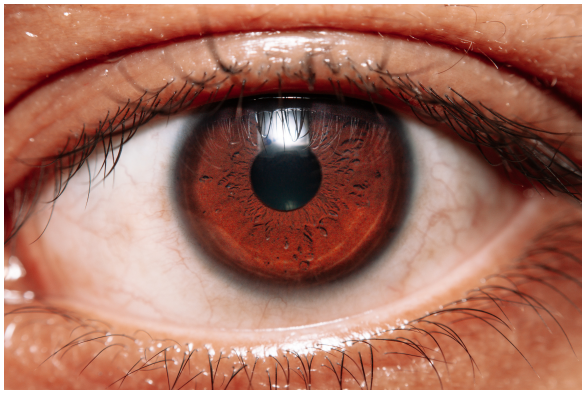

Figure 1.3: Original Image

Two key hyperparameters that define the convolution operation are size and number of kernels. The kernel values for the each layer will be randomly initialized using a uniform distribution within a small range of values. These values are crucial as they determine the filters that the convolutional layer uses to extract features from the input images.

```
1  import numpy as np
2  from keras.models import Sequential
3  from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
4
5  # Initialize the CNN
6  classifier = Sequential()
7
8  # First Convolutional Layer
9  classifier.add(Conv2D(32, (3, 3), activation='relu', input_shape
       =(128, 128, 3), padding='same', name='conv1'))
10 classifier.add(MaxPooling2D(pool_size=(2, 2)))
```

For example: conv layer1 kernels

Kernel 1:

$$\begin{bmatrix} -0.03968783 & -0.06642497 & 0.10930327 \\ -0.02291273 & -0.05866925 & -0.12145701 \\ 0.04586992 & -0.12538332 & 0.00503162 \end{bmatrix}$$

$$\begin{bmatrix} -0.13789758 & 0.10363115 & 0.08985095 \\ -0.09204426 & -0.07554051 & 0.05985981 \\ 0.10929883 & -0.08031537 & 0.13161267 \end{bmatrix}$$

$$\begin{bmatrix} 0.02503161 & -0.1147864 & 0.04518789 \\ 0.02964056 & 0.10101455 & -0.1175285 \\ -0.07683762 & -0.00048338 & 0.13379954 \end{bmatrix}$$

Kernel 2:

$$\begin{bmatrix} 0.12109698 & -0.07206029 & 0.03116728 \\ 0.00921278 & 0.09333126 & 0.02159177 \\ 0.0652937 & 0.12910424 & -0.08964779 \end{bmatrix}$$

$$\begin{bmatrix} 0.03540896 & 0.06898125 & -0.13766396 \\ -0.12430757 & -0.09485874 & -0.00147976 \\ 0.07108943 & 0.02755406 & 0.12318249 \end{bmatrix}$$

$$\begin{bmatrix} -0.06615643 & 0.05311194 & -0.13306625 \\ 0.03425235 & -0.0318674 & 0.07313068 \\ -0.07599039 & -0.13404952 & 0.03824864 \end{bmatrix}$$

...

Kernel 32:

$$\begin{bmatrix} -0.12876226 & 0.1039672 & 0.00147872 \\ -0.09334265 & -0.10519394 & -0.06679485 \\ -0.07421958 & -0.11436825 & 0.1273932 \end{bmatrix}$$

Similarly, Each Convolutional layers will have corresponding number of kernels of sizes specified.

1.1.2 Feature map

An element-wise product between each element of the kernel and the input tensor is calculated at each location of the tensor and summed to obtain the output value in the corresponding position of the output tensor, called *feature map*. This procedure repeated applying multiple kernels to form an arbitrary number of feature maps, which represent different characteristics of the input tensors; different kernels can, thus, be considered as different feature extractors.

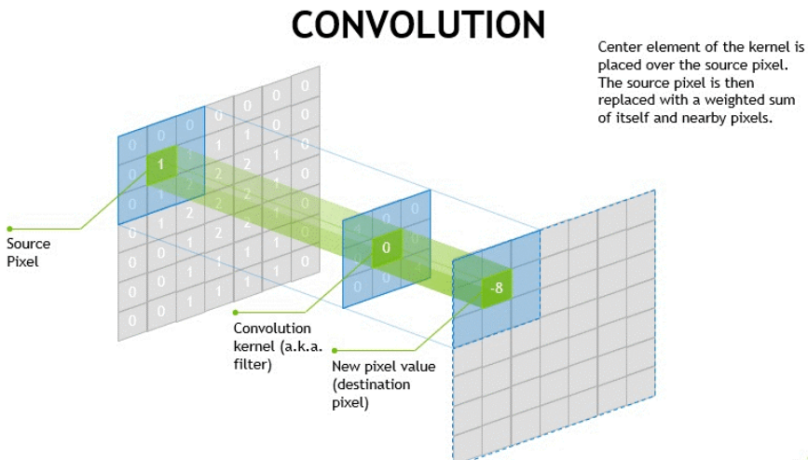

Figure 1.4: A convolutional layer with kernel size of 3x3.

Given an input tensor X and a kernel (filter) tensor K, the convolution operation produces an output tensor Y using the following formula:

$$Y_{i,j} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X_{i+m,j+n} \cdot K_{m,n} + b$$

where:

- $Y_{i,j}$ is the value of the output tensor at position (i, j),
- $X_{i+m,j+n}$ is the value of the input tensor at position $(i + m, j + n)$,
- $K_{m,n}$ is the value of the kernel tensor at position (m, n),
- M and N are the dimensions of the kernel tensor,
- b is the bias term.

And this is passed into the ReLU activation function:

$$Y_{\text{ReLU}_{i,j}} = \text{ReLU}(Y_{i,j})$$

where:

$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

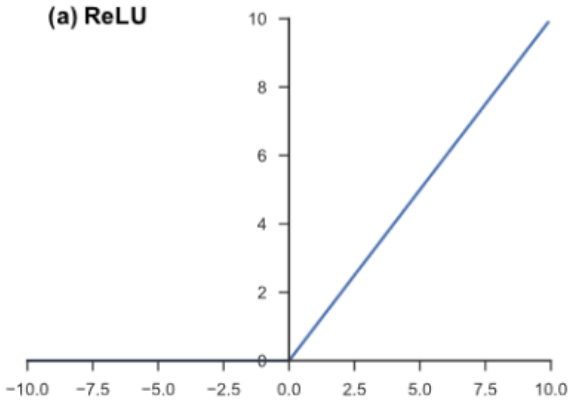

Figure 1.5: ReLU activation function

```
1 # First Convolutional Layer
2 classifier.add(Conv2D(32, (3, 3), activation='relu', input_shape
      =(128, 128, 3), padding='same', name='conv1'))
3 classifier.add(MaxPooling2D(pool_size=(2, 2)))
```

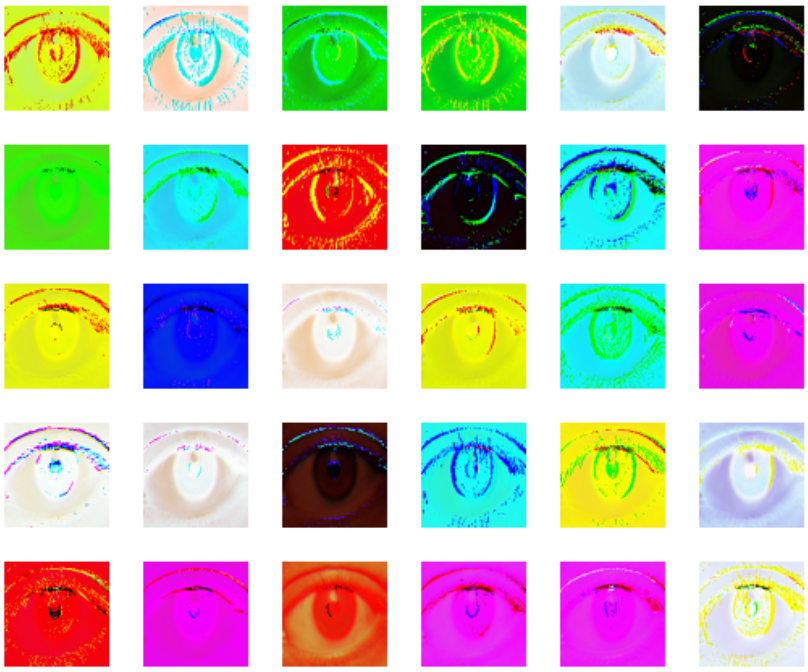

Figure 1.6: a total of 32 distinct patterns are generated as a result of the first convolution layer when the image from Figure 1.1 is fitted into the model.

Valid Padding

Valid padding, specified as `padding='valid'`, means no padding is added around the input. As a result, the output dimensions are reduced after the convolution operation. The output dimension is calculated as:

$$\text{Output Dimension} = (\text{Input Dimension} - \text{Filter Size} + 1)$$

For example, applying a 3×3 filter to a 128×128 input with valid padding results in an output dimension of:

$$128 - 3 + 1 = 126$$

Same Padding

Same padding, specified as `padding='same'`, adds padding such that the output dimensions are the same as the input dimensions. This is achieved by adding enough zeros around the border of the input. The formula for the output dimension with same padding is:

$$\text{Output Dimension} = \left\lceil \frac{\text{Input Dimension}}{\text{Stride}} \right\rceil$$

where the stride is usually 1. For instance, applying a 3×3 filter to a 128×128 input with same padding maintains the output dimension as 128×128.

1.1.3 Pooling

The most popular form of pooling operation is max pooling, which extracts patches from the input feature maps, outputs the maximum value in each patch, and discards all the other values. A max pooling with a filter of size 2×2 with a stride of 2 is commonly used in practice. This downsamples the in-plane dimension of feature maps by a factor of 2. Unlike height and width, the depth dimension of feature maps remains unchanged. The distance between two successive kernel positions is called a stride, which also defines the convolution operation. The common choice of a stride is 1; however, a stride larger than 1 is sometimes used to achieve downsampling of the feature maps.

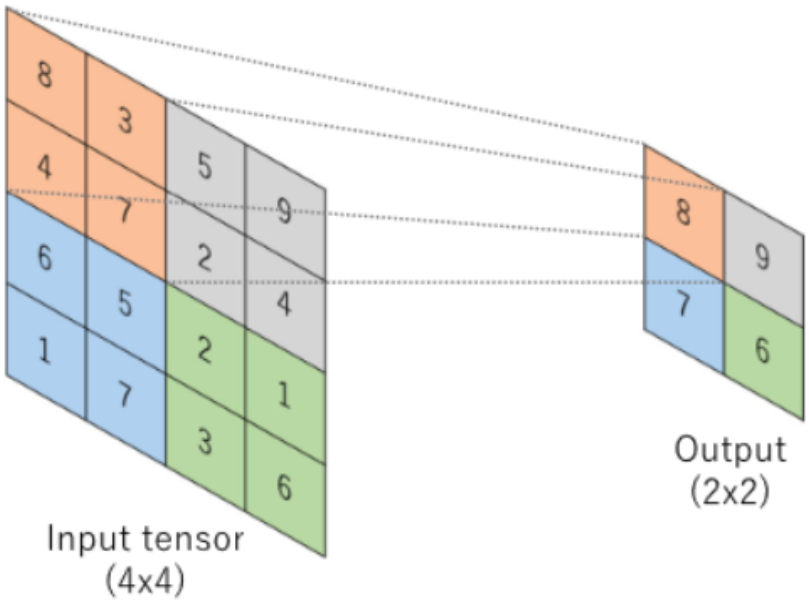

Figure 1.7: Max Pooling

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

# Initialize the CNN
classifier = Sequential()

classifier.add(Conv2D(32, (3, 3), activation='relu', input_shape
    =(128, 128, 3), padding='same', name='conv1'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

# Second Convolutional Layer
classifier.add(Conv2D(64, (3, 3), activation='relu', padding='same'
    , name='conv2'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

# Third Convolutional Layer
classifier.add(Conv2D(128, (3, 3), activation='relu', padding='same
    ', name='conv3'))
```

```
classifier.add(MaxPooling2D(pool_size=(2, 2)))

# Fourth Convolutional Layer
classifier.add(Conv2D(256, (3, 3), activation='relu', padding='same', name='conv4'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))
```

Results of max pooling

Input Layer

- Input Shape: (128, 128, 3)

First Convolutional Layer

- Output Dimensions: 128x128x32 (same as input because of "same" padding)
- After Max Pooling: 64x64x32 (pooling reduces each dimension by 2)

Second Convolutional Layer

- Output Dimensions: 64x64x64
- After Max Pooling: 32x32x64

Third Convolutional Layer

- Output Dimensions: 32x32x128
- After Max Pooling: 16x16x128

Fourth Convolutional Layer

- Output Dimensions: 16x16x256
- After Max Pooling: 8x8x256

1.1.4 Fully Connected Layer

The output feature maps of the final convolution or pooling layer is typically flattened, i.e., transformed into a one-dimensional (1D) array of numbers (or vector), and connected to one or more fully connected layers, also known as dense layers, in which every input is connected to every output by a learnable weight. Once the features extracted by the convolution layers and downsampled by the pooling layers are created, they are mapped by a subset of fully connected layers to the final outputs of the network, such as the probabilities for each class in classification tasks. The final fully connected layer typically has the same number of output nodes as the number of classes.

$$\mathrm{h}_i = \mathrm{ReLU}(b_i + \sum_{i=1}^{16384} x_i \cdot w_i) \quad \text{where} \quad \begin{cases} h_i \text{ represents the output of the } i^{th} \text{ neuron} \\ b_i \text{ is the bias term for the } i^{th} \text{ neuron} \\ x_j \text{ is the } i^{th} \text{ input to the layer} \\ w_i \text{ is the weight connecting the } i^{th} \text{ input to the } i^{th} \text{ neuron} \end{cases}$$

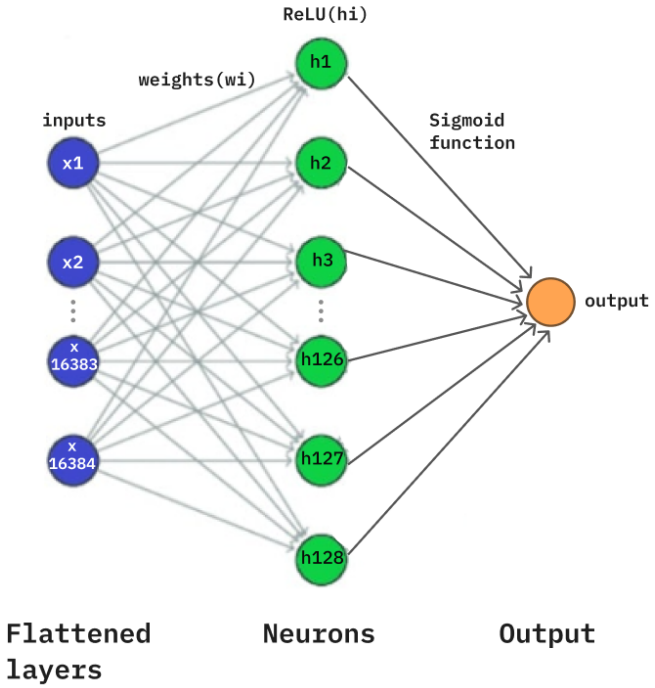

Figure 1.8: Fully Connected Layer

$$\text{Output} = \text{Sigmoid}\left(\sum_{i=1}^{128} h_i\right) \quad \text{where} \quad \begin{cases} \text{SigmoidSigmoid}(x) = \frac{1}{1+e^{-x}} \\ \text{Output is the final output of the network} \end{cases}$$

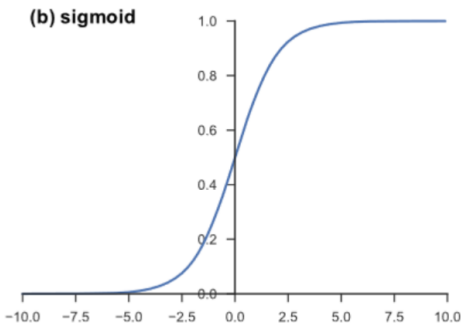

Figure 1.9: Sigmoid Activation function

```
1 import numpy as np
2 from keras.models import Sequential
3 from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
4
5 # Initialize the CNN
```

```
classifier = Sequential()

# First Convolutional Layer
classifier.add(Conv2D(32, (3, 3), activation='relu', input_shape
    =(128, 128, 3), padding='same', name='conv1'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

# Second Convolutional Layer
classifier.add(Conv2D(64, (3, 3), activation='relu', padding='same'
    , name='conv2'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

# Third Convolutional Layer
classifier.add(Conv2D(128, (3, 3), activation='relu', padding='same
    ', name='conv3'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

# Fourth Convolutional Layer
classifier.add(Conv2D(256, (3, 3), activation='relu', padding='same
    ', name='conv4'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))


# Flattening the output into a 1D array
classifier.add(Flatten())

classifier.add(Dense(units=128, activation='relu', name='fcl'))
classifier.add(Dense(units=1, activation='sigmoid', name='output'))

def log_initial_kernels(model):
    for layer in model.layers:
        if layer.name == 'conv1':
            kernels, biases = layer.get_weights()
            print(f"Layer {layer.name} initial kernels:")
            for i in range(kernels.shape[-1]):
                kernel_matrix = kernels[:, :, :, i]
                print(f"Kernel {i+1}:\n{kernel_matrix}\n")

log_initial_kernels(classifier)

# Compiling the CNN
classifier.compile(optimizer='adam', loss='binary_crossentropy',
    metrics=['accuracy'])
```