CHAPTER 37

WORKING PRINCIPLES OF CONVOLUTIONAL NEURAL NETWORKS IN KERAS

Devrim AKGUN[1]

1Sakarya University, Computer and Information Sciences Faculty, Software Engineering Department, ORCID: 0000-0002-0770-599X

Akgun, Devrim, Working Principles Of Convolutional Neural Networks in Keras, Academic Research and Reviews in Engineering Sciences, Platanus Publishing, pp:659-674,2023

1. INTRODUCTION

Deep learning models generally use artificial neural networks as classifiers, consisting of fully connected layers consisting of a varying number of layers depending on the problem type. Classifier inputs, which must be determined manually in machine learning, are automatically obtained from the data through some feature extractor layers in deep learning networks. In other words, during training, these layers are trained to select features that will increase success. Convolutional neural networks (CNN) are among the most commonly preferred networks in deep learning models. CNN is an artificial neural network designed to process data that may be encountered in practice, such as signals, sequences, images, or volumetric data (LeCun et al., 2015). CNN models have proven effective in many applications, such as image classification, object detection, and image recognition.

The most crucial feature of CNN architectures is the use of layers that apply convolution operations with trainable coefficients called kernels on the input data. Convolution involves shifting the kernel over the input data and performing multiplication and addition operations with weights on the input data to create a feature map. Thus, it helps capture local and global relationships by learning important features in the input data.

CNN consist of convolutional and fully connected layers to form a deep learning model for classification and regression (Goodfellow et al. 2016). Convolutional layers in deep learning models provide an excellent feature extraction method for reducing the input data while keeping essential features. TensorFlow provides various convolutional layers, such as *Conv1D* for one-dimensional convolution and *Conv2D* for two-dimensional convolution. These layers proved useful in major applications such as computer vision, audio processing, and sequence processing.

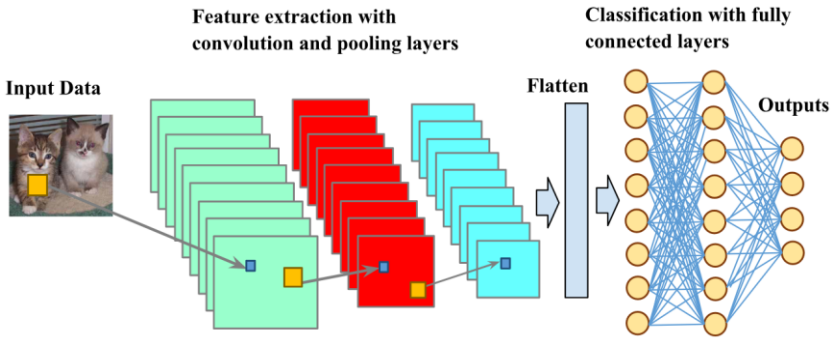


Figure 1. An example convolutional neural network structure

Figure 1 shows a typical CNN structure that can be used for image classification. The structure of the network includes convolution layers, activation functions, pooling layers, and a fully connected layer. In practice, defining and training such a model for a purpose can be accomplished with various deep learning application programming interfaces (APIs). Keras API is one of the widely used tools for this purpose (Chollet 2021).

2. Keras API

Keras is a machine learning Application Programming Interface (API) written in Python that runs on the TensorFlow platform (Chollet 2015). It provides efficient tools for training, analyzing, and testing deep learning models more practically. Layers such as the convolution, pooling, and fully connected layers, which are necessary to realize a typical CNN model, can be easily defined with Keras either using the sequential form or functional form. In addition, it also provides many deep learning tools, such as activation functions, loss functions, optimization algorithms, and performance metrics.

Keras includes various layers like *Conv2D*, which is one of the popular layers for processing two-dimensional tensors like images or two-dimensional features. The *Conv1D* layer processes one-dimensional tensors like word embeddings, signals, or sequences. For example, the one-dimensional features from a network intrusion dataset can be processed using models with *Conv1D* layers. Also, these features can be converted to two-dimensional representation and can be classified with models using *Conv2D* layers (Çavuşoğlu et al. 2023). Both functions have some important common parameters as described below:

- **filters:** The filters parameter defines the number of filters in the layer. For example, 32 filters are used simultaneously in the first layer to extract features from the image.
- **kernel_size:** Each filter's kernel sizes are defined as 3x3 with the kernel_size parameter.
- **activation:** After the convolution process, the extracted features are applied to the activation function for additional nonlinear processing to improve the output. For example, the rectifying linear unit (relu) function rounds the negative values to zero; the sigmoid function limits the output between 0 and 1.
- **padding:** When applying convolution, the number of rows and columns decreases depending on the kernel size when the default parameter *padding*='valid' is used. Selecting *padding*='same' keeps the output shape the same as the input shape by adding some rows and columns of zeros, depending on the kernel size.
- **input_shape / shape:** The number of inputs of each layer is determined by the number of outputs of the previous one. Only in the first layer, the input size needs to be specified according to the dataset's properties. For example, the *input_shape*= (128, 128, 3) shows that the input image is 128x128 and has three channels. In the functional form, a separate input layer with shape parameters is used to take the input.

3. Working principles of Conv2D layers

Keras implements two-dimensional convolutional layers using the *Conv2D* function. As shown by Figure 2, *Conv2D* can simply be included in a model using *keras.layers*. The example model just contains one filter with 3x3 kernel size and no activation function to illustrate the behavior of the *Conv2D* layer. The network input is set to a single channel input with 8x8 dimensions for the example input with randomly generated numbers. The Model class takes the inputs and outputs from the network to form the Keras model. The *model.summary*() prints some model information such as layer types, output shape, and the number of parameters in the network.

```
from keras import Model, layers
import numpy as np

# a test input of size 8x8
in1=layers.Input(shape=(8,8,1))

out1=  layers.Conv2D(filters=1,
                     kernel_size=(3,3),
                     activation=None,
                     padding='valid',
                     )(in1)
# define a Keras model object
model=Model(inputs=in1,outputs=out1)
# print the model information
model.summary()
```

Figure 2. A single Conv2D layer for testing

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 8, 8, 1)]         0

 conv2d (Conv2D)             (None, 6, 6, 1)           10

=================================================================
Total params: 10 (40.00 Byte)
Trainable params: 10 (40.00 Byte)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________

In [2]: W=model.get_weights() # Initial kernel weights

In [3]: print(W[0][:,:,0,0])
[[ 0.18230325  0.27055514 -0.3342654 ]
 [ 0.5403837   0.5592307  -0.06283993]
 [-0.00818723  0.16785127 -0.26100808]]

In [4]: print(W[1]) # Initial bias
[0.]
```

Figure 3. Summary and initial weights of the model

Figure 3 shows the output of *model.summary()* and the initial values of the weights. There are a total of 10 trainable parameters: 9 of them belong to the 3x3 kernel and one for bias. The initial values of the trainable parameters are assigned randomly before training starts. We can get the current values of the trainable parameters using *get_weights()* function of the model. The output shape of the layer is decreased from 8x8 to 6x6 since we set *padding='valid'*.

```
from numpy import array
# set weights manually
W=[array([[[[0.0]],  [[1.0]],  [[0.0 ]]],
          [[[1.0]],  [[-4.0]], [[1.0]]],
          [[[0.0]],  [[1.0]],  [[0.0]]]], dtype='float32'),
   array([0.], dtype='float32')]

model.set_weights(W)

#sample input data
sample_in=np.random.randint(0,20,size=(8,8)).astype('float32')

# run the model with single convolution
y=model.predict(sample_in.reshape(1,8,8,1))
```

Figure 4. Applying a single convolution to an input image of 8x8

Network parameters can be assigned manually using *the set_weights()* function of the model. This is useful for loading pre-trained weights in transfer learning applications where weights are transferred from another network (Krishna et al. 2019). In this simple experiment given by Figure 4, after weights are loaded manually using predefined kernel values, the randomly generated input is applied to the model using the *predict()* function. The outputs from the predict() function, the randomly generated input, and an example operation on the selected window are displayed in Figure 5. The reduction in the output dimensions can be compensated by setting *padding='same'*. This parameter pads input with zeros based on the kernel size, as illustrated by Figure 6, where the output shape equals the input shape. Therefore, the padded lines are the eliminated lines, and therefore, the *padding='same'* parameter maintains the input shape at the output.

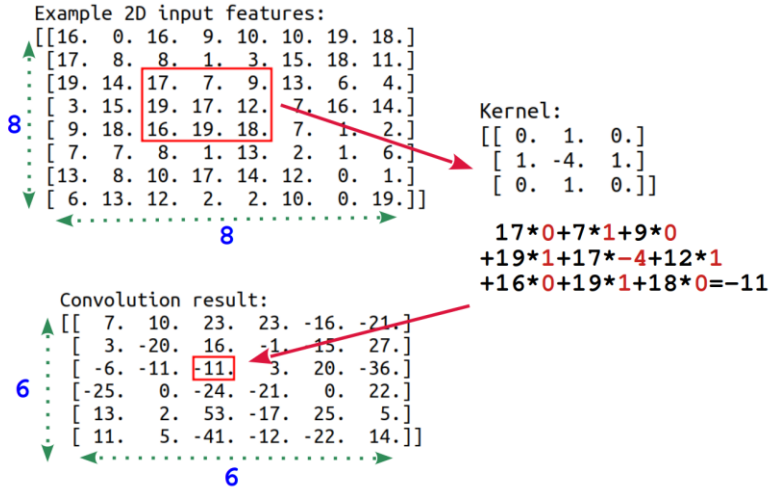


Figure 5. Illustration of convolution on random input features

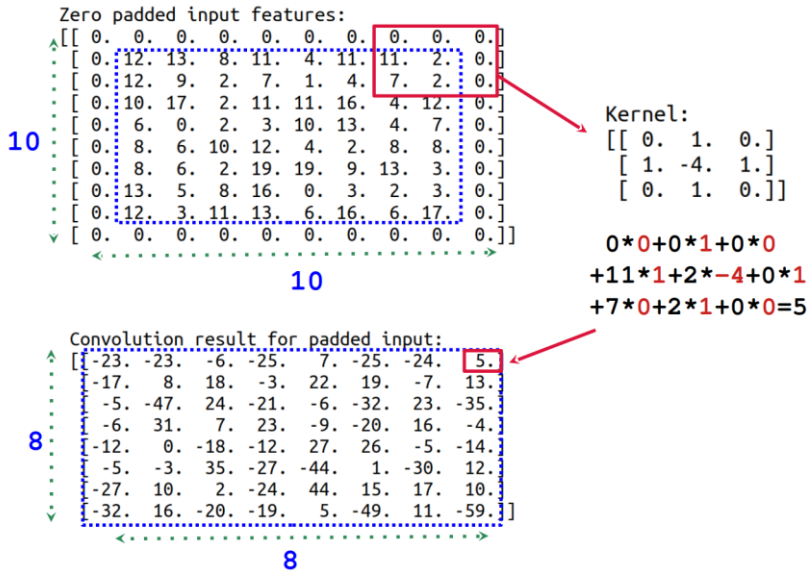


Figure 6. Illustration of convolution with zero padding on random input features

4. A network using Conv2D layers

The example operations above illustrate the basic working principles of a single convolution operation for a given feature map. Conv2D enables multiple kernels simultaneously so that each kernel can extract another useful feature for the deep learning model. Figure 7 illustrates a simple convolutional layer with just three filter units. Because input includes just a single channel, three kernels

are defined for processing the input. In the case of a color image or multiple-channel data, the number of kernels is multiplied by the number of channels. Figure 8 illustrates connections for color input where 3-three channels for red, green, and blue images. In this case, the number of convolutions is increased from 3 to 9. The number of outputs for the next layer is three, either for single or multiple-channel cases. The following layer takes the outputs as inputs, and therefore, the number of inputs is determined automatically. This means there are three inputs for the following layer, as shown in Figure 9, where the connections are illustrated for an interval layer. It is like a fully connected layer with three inputs and three outputs, but each connection represents a convolution operation. For this example, there are a total of 9 convolutions.

```
x=Conv2D(filters=3,
         kernel_size=(3,3),
         activation='relu',
         padding='same')(in)
```

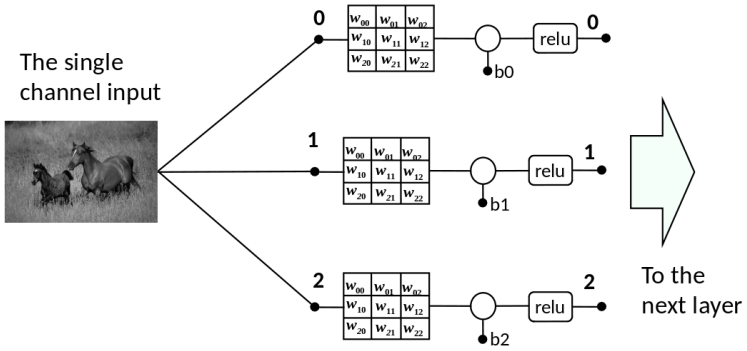


Figure 7. Convolutional input layer for a single channel image

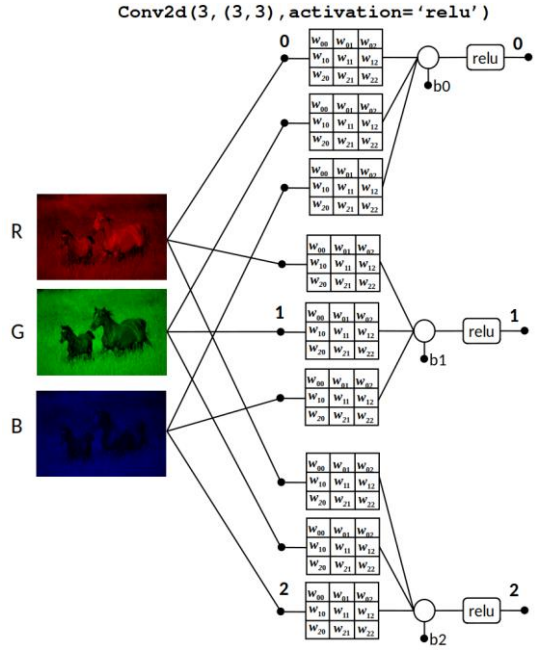


Figure 8. Convolutional input layer for a 3-channel image

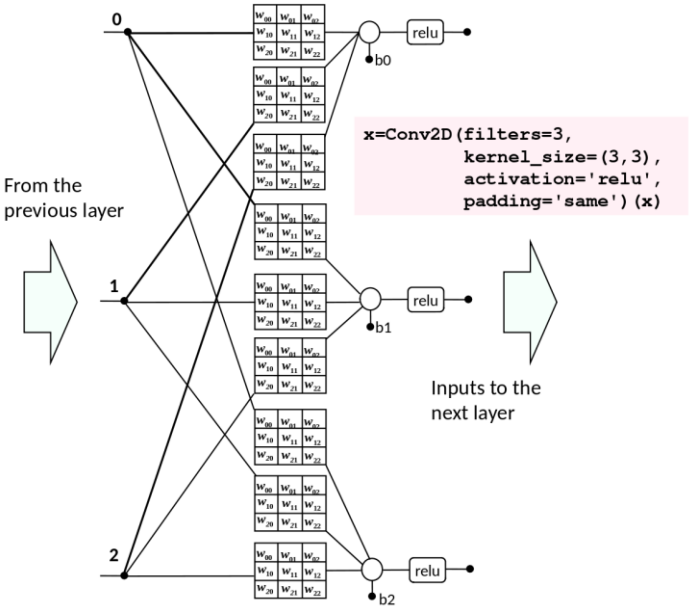


Figure 9: *A* convolutional interval layer example

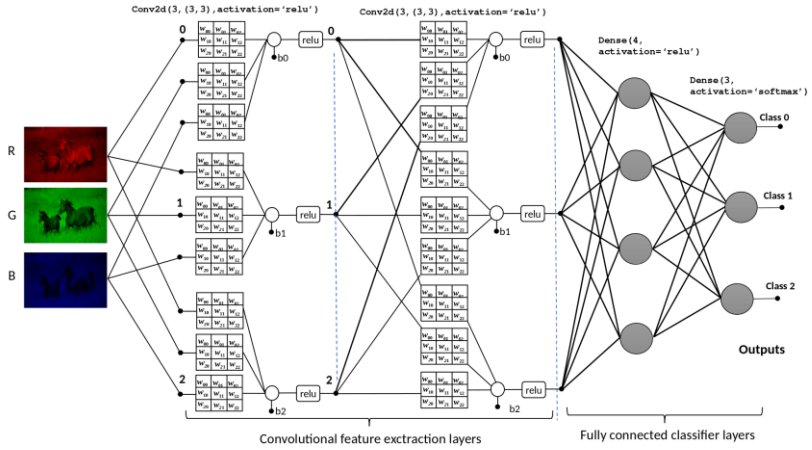


Figure 10. A simple convolutional neural network to illustrate connections

Figure 10 represents a simple, complete model with fully connected layers. A dense layer defines the fully connected layers with a specified number of units. Figure 11 represents the full Keras definition of the whole model.

```
in1=layers.Input(shape=(8,8,1))
# The first convolutional layer
x=  layers.Conv2D(filters=3,
                  kernel_size=(3,3),
                  activation='relu',
                  padding='same',
                  )(in1)
# The second convolutional layer
x=  layers.Conv2D(filters=3,
                  kernel_size=(3,3),
                  activation='relu',
                  padding='same',
                  )(x)
# Flatten the outputs for Dense layer
x=  layers.Flatten()(x)
# The fully connected layers
x=layers.Dense(4, activation='relu')(x)
out1=layers.Dense(3, activation='softmax')(x)

# define model
model=Model(inputs=in1,outputs=out1)

#print summary of the model
model.summary()
```

Figure 11. Keras model definition of the model given by Figure 10

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 input_1 (InputLayer)         [(None, 8, 8, 1)]         0

 conv2d (Conv2D)              (None, 8, 8, 3)           30

 conv2d_1 (Conv2D)            (None, 8, 8, 3)           84

 flatten (Flatten)            (None, 192)               0

 dense (Dense)                (None, 4)                 772

 dense_1 (Dense)              (None, 3)                 15

=================================================================
Total params: 901
Trainable params: 901
Non-trainable params: 0
```

Figure 12. Summary of the example model

Figure 12 displays the summary of the full model, which was produced with *model.summary()*. There are a total of 901 trainable parameters in the whole model, and they vary according to the type of layer. For example, the number of parameters (*nparams*) for a convolutional layer is equal to the multiplication of the number of inputs (*ninputs*) and the number of units (*units*) and the square of the kernel_size. The number of biases is equal to the number of units as given by Eq. (1).

$$nparams=ninputs*units*kernel\_size*kernel\_size+units \quad (1)$$

- For the first Convolutional layer:

nparams=1*3*3*3+3=30

- For the second Convolutional layer:

nparams=3*3*3*3+3=84

The size of the flattened vector is equal to the number of convolutions and their dimensions. For this example, there are three convolutions for the second layer, each containing 8x8 features. Therefore, the size of the flatten is 8*8*3=192. This means the fully connected layer has 192 inputs, and because there are four units for the first dense layer, the number of trainable parameters together with biases per unit is 192*4+4=772. For the second fully connected layer, the number of inputs equals the number of outputs from the previous layer. Therefore, there are a total of 4*3+3=15 trainable parameters together with the number of biases.

```
def cnn_model():
    in1=layers.Input(shape=(32, 32, 3))

    x=layers.Conv2D(filters=32,
                            kernel_size=(3,3),
                            activation='relu',
                            padding='same')(in1)
    x=layers.MaxPool2D()(x)
    x=layers.Conv2D(filters=64,
                            kernel_size=(3,3),
                            padding='same',
                            activation='relu')(x)
    x=layers.MaxPool2D()(x)
    x=layers.Conv2D(filters=64,
                            kernel_size=(3,3),
                            padding='same',
                            activation='relu')(x)
    x=layers.MaxPool2D()(x)
    x=layers.Flatten()(x)
    x=layers.Dense(512,activation='relu')(x)
    out1=layers.Dense(10,activation='softmax')(x)
    # define model
    model=Model(inputs=in1,outputs=out1)
    model.summary()
    return model
```

Figure 13. A typical convolutional neural network for multiclass classification

Figure 13 illustrates a larger model than our simple example for multiclass classification of input images. This model takes 3-channel images with 32x32 dimensions and classifies them into ten classes, as indicated by the number of units in the last layer. This layer also includes the *MaxPool2D* pooling layers for dimension reduction. By default, the pooling size is 2x2, and if a pooling operation follows the output, the output shapes are reduced by half the size, as illustrated in the summary of the model given in Figure 14. For example, the input size for the first MaxPool2D is 32x32, as indicated by the output shape of the first Conv2D layer. Therefore, the output shape of the MaxPool2D leads to a 16x16 shape. Therefore, it reduces the shape of the features while keeping the important ones. The output for the last MaxPool2D layer reduces to 4x4, making it suitable for the fully connected layer.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 32, 32, 3)]       0

 conv2d_2 (Conv2D)           (None, 32, 32, 32)        896

 max_pooling2d (MaxPooling2D  (None, 16, 16, 32)       0
 )

 conv2d_3 (Conv2D)           (None, 16, 16, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 64)         0
 2D)

 conv2d_4 (Conv2D)           (None, 8, 8, 64)          36928

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten_1 (Flatten)         (None, 1024)              0

 dense_2 (Dense)             (None, 512)               524800

 dense_3 (Dense)             (None, 10)                5130

=================================================================
Total params: 586,250
Trainable params: 586,250
Non-trainable params: 0
```

Figure 14. Summary of the model *cnn_model()*

5. Working principles of Conv1D layers

One-dimensional convolutions work like two-dimensional convolutions, except they process one-dimensional features, as described in Figure 15. Conv1D shares parameters set similar to the Conv2D layer, such as kernel_size, padding, and strides. Figure 16 shows an example model with a single convolutional layer where the weights are set with specific numbers to check the working principles. Figure 16 shows the example kernel values, input features, and the computed output for the single kernel experiment. The numerical values are straightforward to verify the results. Figure 18 also repeats the results for random weights and two Conv1D units. In this case, there are two kernels and, therefore, two output feature arrays for the next layer.

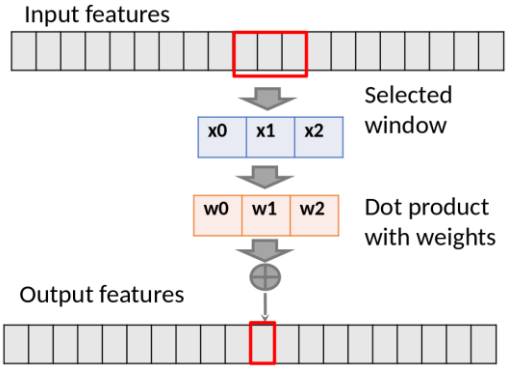


Figure 15. Illustration of Conv1D for kernel_size=3

```
from keras import layers, Model
import numpy as np

sample_in=np.random.randint(1,10,size=(10,1)).astype('float32')
input1=layers.Input(shape=(10,1))
conv1=layers.Conv1D(1,
                    3,
                    strides=1,
                    padding='valid',
                    activation='relu')(input1)
model=Model(inputs=input1,outputs=conv1)
model.summary()

W=[np.array([[[2.0]],
             [[1.0]],
             [[3.0]]]),
np.array([0.])]

model.set_weights(W)

y=model.predict(sample_in.reshape(1,10,1))

print("\nKernel:")
print(W[0][:,0,0])

print("\nExample 1D input features:")
print(sample_in[:,0])

print("\nConvolution result:")
print(y[0,:,0])
```

Figure 16. Conv1D layer example

```
Kernel:
[2. 1. 3.]

Example 1D input features:
[2. 3. 6. 9. 2. 1. 5. 1. 2. 3.]

Convolution result:
[25. 39. 27. 23. 20. 10. 17. 13.]
```

Figure 17. Example processing results using Conv1D

Similarly, the number of inputs for the Conv1D layer can be increased to process multiple features. Figure 17 shows example results where the Conv1D layer has two units with random weights. Note that there is a reduction in the shape as a result of convolution, and this can be fixed by setting the padding= 'same' as in the Conv2D layer.

```
Kernel 1:
[0.4928695  0.17138624 0.77633095]

Kernel 2:
[0.24819446 0.7414068  0.39970565]

Example 1D input features:
[6. 9. 4. 8. 9. 9. 7. 4. 1. 7.]

Convolution result for output 1:
[ 7.6050167 11.332018  10.329546  12.47241   11.412619   8.740852
  4.9119625  7.577181 ]

Convolution result for output 2
[ 9.760651   8.397022  10.521383  12.255568  11.70435    9.022421
  5.1026945  4.532124 ]
```

Figure 17. Results for two Conv1D units with random weights

6. Conclusions

Building CNN-based models with Keras API has become quite practical. After the desired layers are selected and the model is created, the model becomes ready for training and testing to solve a problem. This work investigated the properties of convolutional layers in Keras using detailed examples of the structure and behavior of Conv1D and Conv2D layers. The relationship between the number of units and the number of trainable parameters of the CNN models has been explained with comparative examples. The basic building blocks like Conv1D, Conv2D, and several other layers provide a practical way to build sophisticated problems for various engineering and scientific solutions.

REFERENCES

Chollet, F. (2015). keras. Retrieved December 15, (2023 https://keras.io)

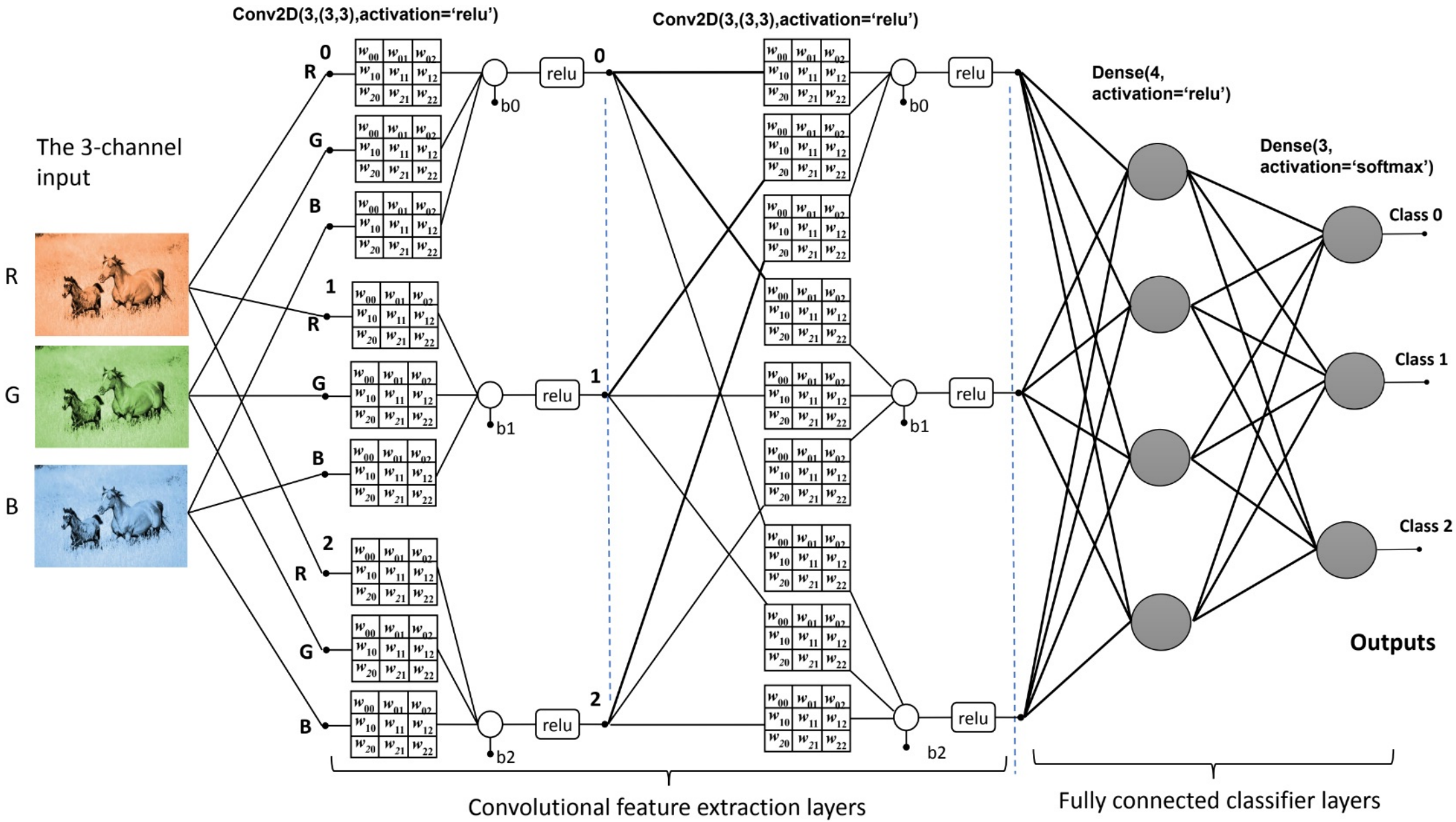
Chollet, F. (2021). Deep Learning with Python, Second Edition. Simon and Schuster.

Çavuşoğlu, Ü., Akgun, D., and Hizal, S. (2023). A Novel Cyber Security Model Using Deep Transfer Learning. Arabian Journal for Science and Engineering, 1-10.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. MIT press.

Krishna, S. T., & Kalluri, H. K. (2019). Deep learning and transfer learning approaches for image classification. International Journal of Recent Technology and Engineering (IJRTE), 7(5S4), 427-432.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.


Conv2D(3,(3,3),activation='relu')
Conv2D(3,(3,3),activation='relu')
Dense(4, activation='relu')
Dense(3, activation='softmax')
The 3-channel input
R
G
B
0
1
2
b0
b1
b2
relu
Class 0
Class 1
Class 2
Outputs
Convolutional feature extraction layers
Fully connected classifier layers