**Experiment 7**: **Implement and Test CNN for Object Detection.**

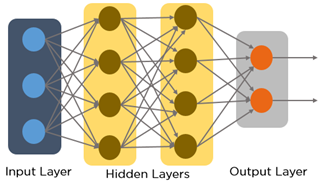
|  |  |
| --- | --- |
| **C409A 5:** | Design and implement convolution neural networks in recognition applications and compare the results obtained using deep learning approach with traditional neural networks. |
| **C409A.6:** | Carry out experiments as an individual and in a team, comprehend and write a laboratory record and draw conclusions at a technical level. |

**SOFTWARES REQUIRED:** MATLAB 7.0 or Python

**THEORY:**

**Neural Networks** is a **generic term in Deep Learning** that works on the basis of the structure and functions of a human brain. Like the human brain has interconnected neurons that constantly transmit signals, a neural network also has interconnected artificial neurons that transmit data among each other and are called as nodes.

A typical neural network consists of 3 layers - input layer, hidden layers and output layer. This is how it looks:

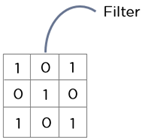
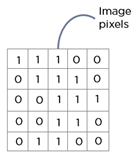


Input layer accepts inputs in different forms. Hidden layers transform the inputs and do several calculations and feature extractions. The output layer produces the desired output. Each node in the network consists of certain random weights and each layers has a bias attached to it that influence the output of every node. Certain activation functions are applied to each layer to decide which nodes to fire.

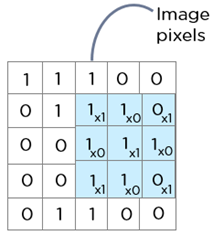
A CNN accepts arrays of pixel values as input to the network. The hidden layer consists of several different layers which carry out feature extraction. There is a fully connected layer that recognizes the objects in the image.

Convolution operation forms the core of every convolution neural network. There are 4 layers in a CNN. These are **Convolution layer, ReLU layer, Pooling layer and Fully Connected Layer**.

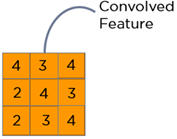
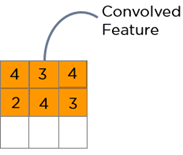
The **Convolution layer** uses a filter matrix over the array of image pixels and performs convolution operation to obtain a **convolved feature map**. Below is an example which represents the convolution operation over the input array.



We shall slide this filter matrix over the input image and compute the convolution operation.

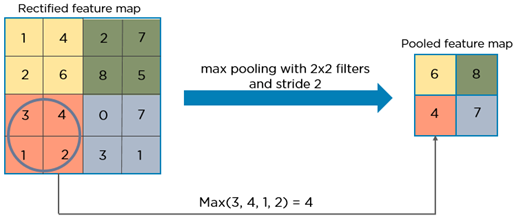


The result is a matrix called the Convolved feature map.



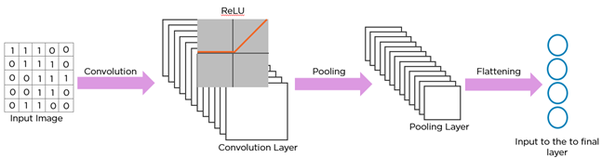
The next layer is the**ReLU layer** which introduces **non-linearity to the network**. It sets all negative pixels to zero and performs element wise operation. The original image is scanned in multiple Convolution and ReLU layers for locating hidden features and patterns in the image. The output is a **Rectified Feature Map**.

The third layer is known as **Pooling layer**. It reduces the dimensionality of the feature map. The output is a **Pooled feature map**.



Pooling layers uses different filters to identify different parts of the image like edges, corners, body, etc.

The pooled feature map is then converted into a **long continuous linear vector**. This process is called **Flattening**. This flattened matrix goes through a Fully Connected Layer to classify the images.



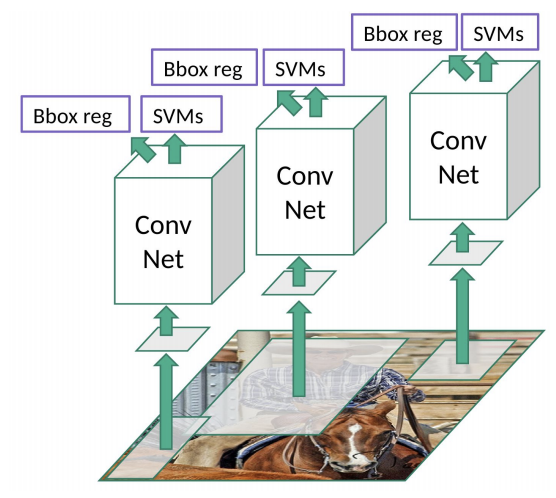
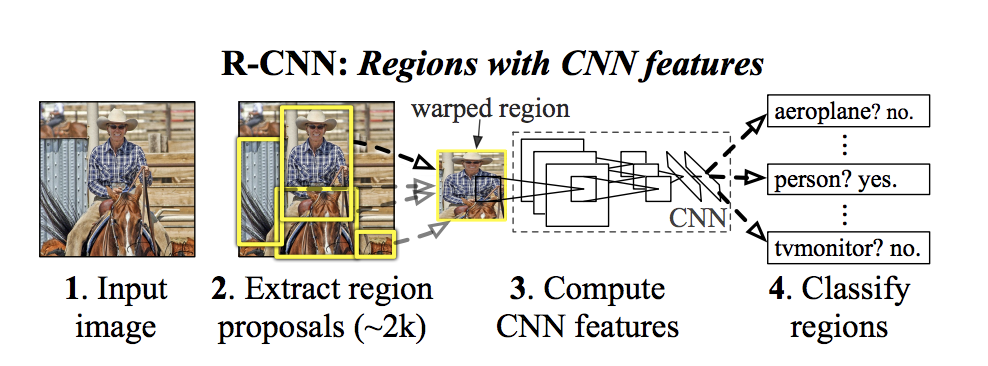
**Limitations of CNN**

The major reason why you cannot proceed with this problem by building a standard convolutional network followed by a fully connected layer is that, the length of the output layer is variable — not constant, this is because the number of occurrences of the objects of interest is not fixed. A naive approach to solve this problem would be to take different regions of interest from the image, and use a CNN to classify the presence of the object within that region. The problem with this approach is that the objects of interest might have different spatial locations within the image and different aspect ratios. Hence, you would have to select a huge number of regions and this could computationally blow up. Therefore, algorithms like R-CNN, YOLO etc have been developed to find these occurrences and find them fast.

R-CNN

To bypass the problem of selecting a huge number of regions, [Ross Girshick et al](https://arxiv.org/pdf/1311.2524.pdf). proposed a method where we use selective search to extract just 2000 regions from the image and he called them region proposals. Therefore, now, instead of trying to classify a huge number of regions, you can just work with 2000 regions. These 2000 region proposals are generated using the selective search algorithm which is written below.

Selective Search:  
1. Generate initial sub-segmentation, we generate many candidate regions  
2. Use greedy algorithm to recursively combine similar regions into larger ones   
3. Use the generated regions to produce the final candidate region proposals

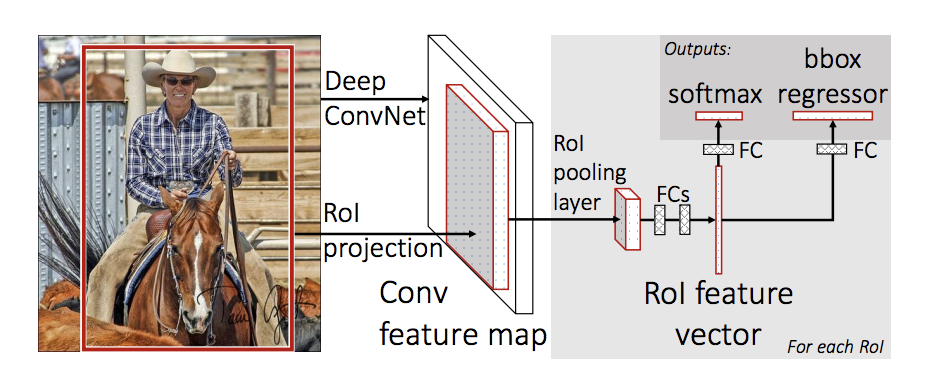


R-CNN

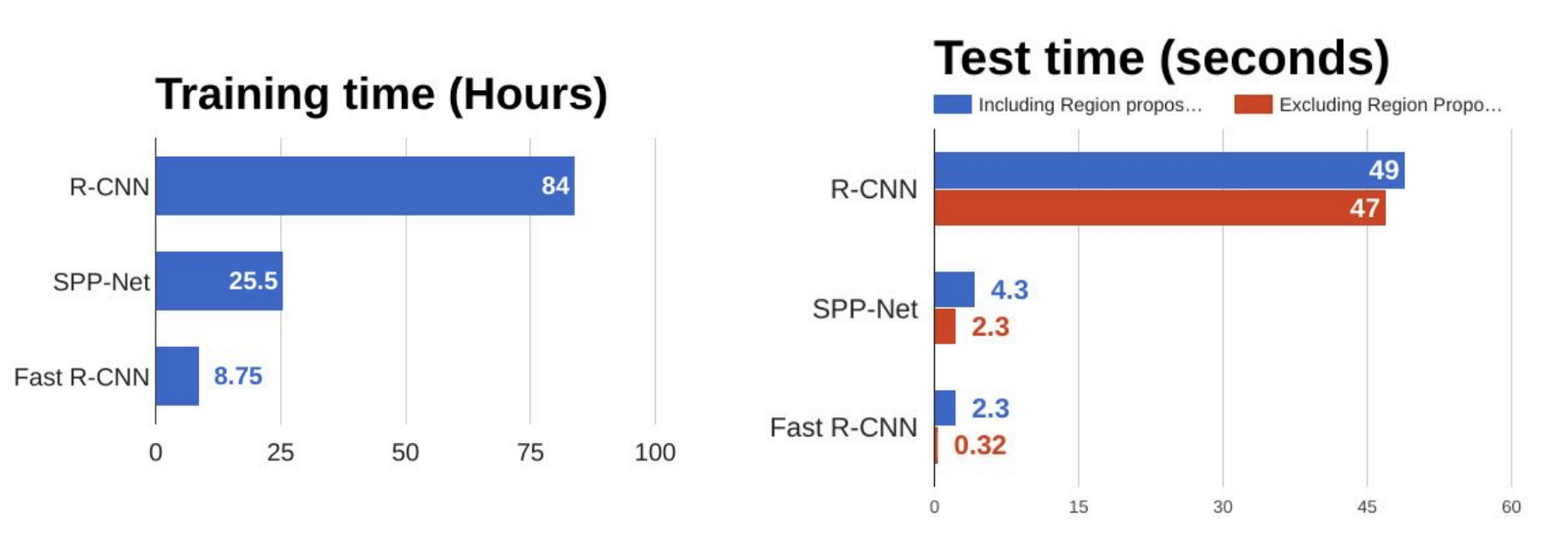
**Limitations of R-CNN**

1. It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
2. It cannot be implemented real time as it takes around 47 seconds for each test image.
3. The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

Fast R-CNN



The same author of the previous paper(R-CNN) solved some of the drawbacks of R-CNN to build a faster object detection algorithm and it was called Fast R-CNN. The approach is similar to the R-CNN algorithm. But, instead of feeding the region proposals to the CNN, we feed the input image to the CNN to generate a convolutional feature map. From the convolutional feature map, we identify the region of proposals and warp them into squares and by using a RoI pooling layer we reshape them into a fixed size so that it can be fed into a fully connected layer. From the RoI feature vector, we use a softmax layer to predict the class of the proposed region and also the offset values for the bounding box.The reason “Fast R-CNN” is faster than R-CNN is because you don’t have to feed 2000 region proposals to the convolutional neural network every time. Instead, the convolution operation is done only once per image and a feature map is generated from it.



# Comparison of object detection algorithms

From the above graphs, you can infer that Fast R-CNN is significantly faster in training and testing sessions over R-CNN. When you look at the performance of Fast R-CNN during testing time, including region proposals slows down the algorithm significantly when compared to not using region proposals. Therefore, region proposals become bottlenecks in Fast R-CNN algorithm affecting its performance.

**Algorithm:**

1. Load vehicle data set 'faster RCNN Vehicle Training Data. mat'

2. Display first few rows of the data set.

3. Split data into a training and test set.

4. Create a Convolutional Neural Network (CNN)

* Create an image input layer.
* Define the convolutional layer parameters.
* Create the middle layers.
* Add a fully connected layer with 64 output neurons.
* The output size of this layer will be an array with a length of 64.
* Add a ReLU non-linearity.
* Add the last fully connected layer

5. Configure Training Options

6. Train Faster R-CNN

7. Evaluate Detector Using Test Set

**RESULTS :**



Fig : Before object detection

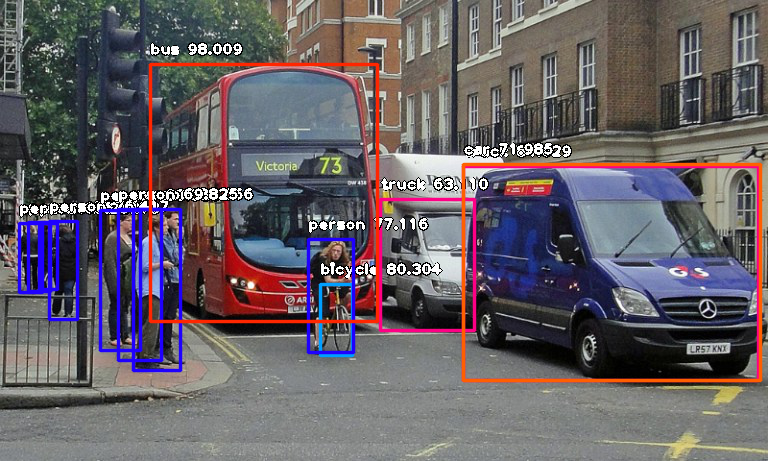


Fig **:** After object detection

**Advantages of Python over MATLAB**

1. Python code tends to be more compact and readable than MATLAB code
2. Python uses zero biased indexing
3. Python offers excellent support for dictionaries(hashes)
4. Python is simple and elegant
5. Python is free and open
6. Any number of functions can be packaged in one file(module)
7. Python’s import statement
8. Python offers more choices in graphics packages and toolsets

**Conclusion:**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**CODE:**

from keras.datasets import cifar10

from keras.utils import np\_utils

from matplotlib import pyplot as plt

import numpy as np

from PIL import Image

(X\_train, y\_train), (X\_test, y\_test) = cifar10.load\_data()

print("Roll No.4855,CNN for object detection")

print('Training Images: {}'.format(X\_train.shape))

print('Testing Images: {}'.format(X\_test.shape))

print(X\_train[0].shape)

for i in range(0,9):

plt.subplot(330 + 1 + i)

img = X\_train[i]

plt.imshow(img)

plt.show()

seed = 6

np.random.seed(seed)

(X\_train, y\_train), (X\_test, y\_test) = cifar10.load\_data()

X\_train = X\_train.astype('float32')

X\_test = X\_test.astype('float32')

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

print(y\_train.shape)

print(y\_train[0])

Y\_train = np\_utils.to\_categorical(y\_train)

Y\_test = np\_utils.to\_categorical(y\_test)

num\_classes = Y\_test.shape[1]

print(Y\_train.shape)

print(Y\_train[0])

from keras.models import Sequential

from keras.layers import Dropout, Activation, Conv2D, GlobalAveragePooling2D

from keras.optimizers import SGD

def allcnn(weights=None):

# define model type - Sequential

model = Sequential()

model.add(Conv2D(96, (3, 3), padding = 'same', input\_shape=( 32, 32,3)))

model.add(Activation('relu'))

model.add(Conv2D(96, (3, 3), padding = 'same'))

model.add(Activation('relu'))

model.add(Conv2D(96, (3, 3), padding = 'same', strides = (2,2)))

model.add(Dropout(0.5))

model.add(Conv2D(192, (3, 3), padding = 'same'))

model.add(Activation('relu'))

model.add(Conv2D(192, (3, 3), padding = 'same'))

model.add(Activation('relu'))

model.add(Conv2D(192, (3, 3), padding = 'same', strides = (2,2)))

model.add(Dropout(0.5))

model.add(Conv2D(192, (3, 3), padding = 'same'))

model.add(Activation('relu'))

model.add(Conv2D(192, (1, 1), padding = 'valid'))

model.add(Activation('relu'))

model.add(Conv2D(10, (1, 1), padding = 'valid')

model.add(GlobalAveragePooling2D())

model.add(Activation('softmax'))

if weights:

model.load\_weights(weights)

return model

learning\_rate = 0.01

weight\_decay = 1e-6

momentum = 0.9

weights = 'all\_cnn\_weights\_0.9088\_0.4994.hdf5'

model = allcnn(weights)

sgd = SGD(lr=learning\_rate, decay=weight\_decay, momentum=momentum, nesterov=True)

model.compile(loss='categorical\_crossentropy', optimizer=sgd, metrics=['accuracy'])

print (model.summary())

scores = model.evaluate(X\_test, Y\_test, verbose=1)

print("Accuracy: %.2f%%" % (scores[1]\*100))

**OUTPUT:**

**CNN for object detection**

**Training Images: (50000, 32, 32, 3)**

**Testing Images: (10000, 32, 32, 3)**

**(32, 32, 3)**

****

**(50000, 1)**

**[6]**

**(50000, 10)**

**[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]**

**Model: "sequential\_2"**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Layer (type) Output Shape Param #**

**=================================================================**

**conv2d\_10 (Conv2D) (None, 32, 32, 96) 2688**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**activation\_8 (Activation) (None, 32, 32, 96) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**conv2d\_11 (Conv2D) (None, 32, 32, 96) 83040**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**activation\_9 (Activation) (None, 32, 32, 96) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**conv2d\_12 (Conv2D) (None, 16, 16, 96) 83040**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dropout\_3 (Dropout) (None, 16, 16, 96) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**conv2d\_13 (Conv2D) (None, 16, 16, 192) 166080**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**activation\_10 (Activation) (None, 16, 16, 192) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**conv2d\_14 (Conv2D) (None, 16, 16, 192) 331968**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**activation\_11 (Activation) (None, 16, 16, 192) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**conv2d\_15 (Conv2D) (None, 8, 8, 192) 331968**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**dropout\_4 (Dropout) (None, 8, 8, 192) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**conv2d\_16 (Conv2D) (None, 8, 8, 192) 331968**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**activation\_12 (Activation) (None, 8, 8, 192) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**conv2d\_17 (Conv2D) (None, 8, 8, 192) 37056**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**activation\_13 (Activation) (None, 8, 8, 192) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**conv2d\_18 (Conv2D) (None, 8, 8, 10) 1930**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**global\_average\_pooling2d\_2 ( (None, 10) 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**activation\_14 (Activation) (None, 10) 0**

**=================================================================**

**Total params: 1,369,738**

**Trainable params: 1,369,738**

**Non-trainable params: 0**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**None**

**10000/10000 [==============================] - 76s 8ms/step**

**Accuracy: 90.88%**