towards
data science

••• Upgrade

Follow 566K Followers · Editors' Picks Features Deep Dives Grow Contribute About

Convolutional Neural Network

Learn Convolutional Neural Network from basic and its implementation in Keras

dshahid380 Feb 25, 2019 · 9 min read

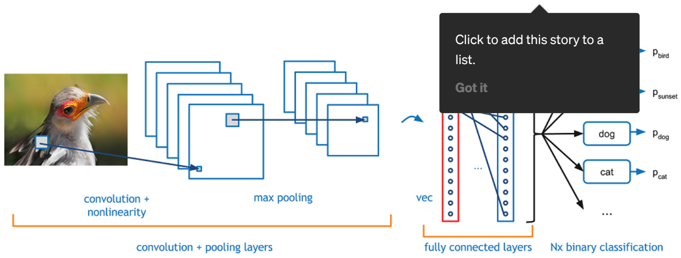


Table of contents

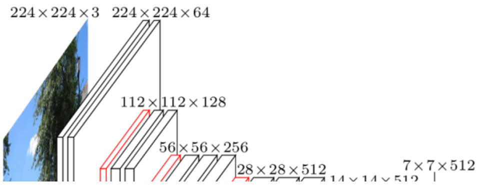
1. What is CNN ?

Computer vision is evolving rapidly day-by-day. Its one of the reason is deep learning. When we talk about computer vision, a term convolutional neural network(abbreviated as CNN) comes in our mind because CNN is heavily used here. Examples of CNN in computer vision are face recognition, image classification etc. It is similar to the basic neural network. CNN also have learnable parameter like neural network i.e, weights, biases etc.

2. Why should we use CNN ?

Problem with Feedforward Neural Network

Suppose you are working with MNIST dataset, you know each image in MNIST is 28 x 28 x 1(black & white image contains only 1 channel). Total number of neurons in input layer will 28 x 28 = 784, this can be manageable. What if the size of image is 1000 x 1000 which means you need 10^6 neurons in input layer. Oh! This seems a huge number of neurons are required for operation. It is computationally ineffective right. So here comes Convolutional Neural Network or CNN. In simple word what CNN does is, it extract the feature of image and convert it into lower dimension without loosing its characteristics. In the following example you can see that initial the size of the image is 224 x 224 x 3. If you proceed without convolution then you need 224 x 224 x 3 = 100, 352 numbers of neurons in input layer but after applying convolution you input tensor dimension is reduced to 1 x 1 x 1000. It means you only need 1000 neurons in first layer of feedforward neural network.



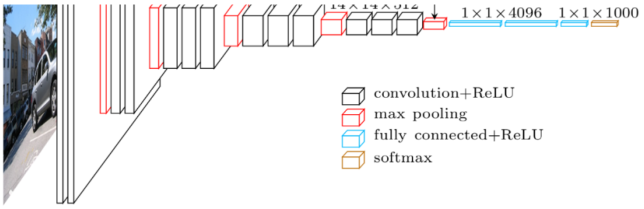


Fig. Downsampling

3. Few Definitions

There are few definitions you should know before understanding CNN

3.1 Image Representation

Thinking about images, its easy to understand that it has a height and width, so it would make sense to represent the information contained in it with a two dimensional structure (a matrix) until you remember that images have colors, and to add information about the colors, we need another dimension, and that is when Tensors become particularly helpful.

Images are encoded into color channels, the image data is represented into each color intensity in a color channel at a given point, the most common one being RGB, which means Red, Blue and Green. The information contained into an image is the intensity of each channel color into the width and height of the image, just like this

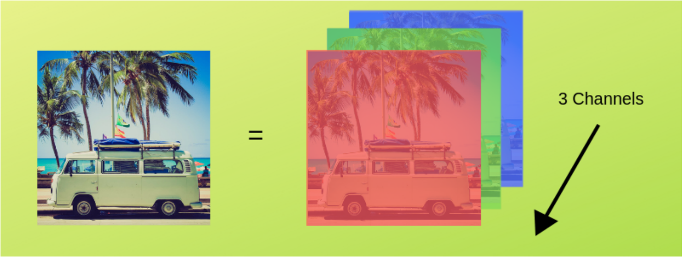
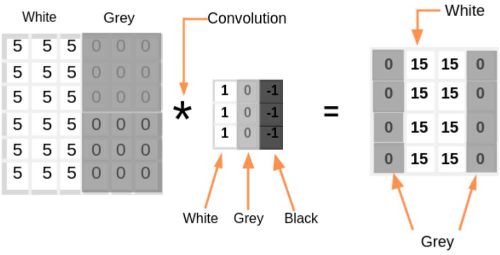


Fig. RGB representation of a image

So the intensity of the red channel at each point with width and height can be represented into a matrix, the same goes for the blue and green channels, so we end up having three matrices, and when these are combined they form a tensor.

3.2 Edge Detection

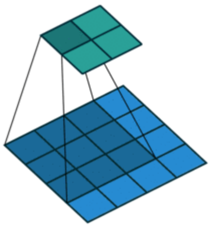
Every image has vertical and horizontal edges which actually combining to form a image. Convolution operation is used with some filters for detecting edges. Suppose you have gray scale image with dimension 6 x 6 and filter of dimension 3 x 3(say). When 6 x 6 grey scale image convolve with 3 x 3 filter, we get 4 x 4 image. First of all 3 x 3 filter matrix get multiplied with first 3 x 3 size of our grey scale image, then we shift one column right up to end , after that we shift one row and so on.



$$\begin{array}{cccccc} 5 & 5 & 5 & 0 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 & 0 \end{array} * \begin{array}{ccc} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{array} = \begin{array}{cccc} 0 & 15 & 15 & 0 \\ 0 & 15 & 15 & 0 \\ 0 & 15 & 15 & 0 \\ 0 & 15 & 15 & 0 \end{array}$$

Convolution operation

The convolution operation can be visualized in the following way. Here our image dimension is 4 x 4 and filter is 3 x 3, hence we are getting output after convolution is 2 x 2.



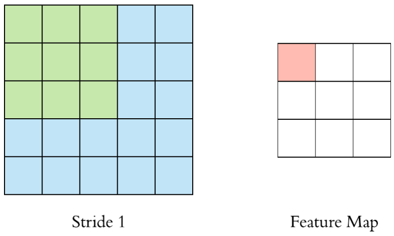
Visualization of convolution

If we have N x N image size and F x F filter size then after convolution result will be

```
(N x N) * (F x F) = (N-F+1)x(N-F+1)(Apply this for above case)
```

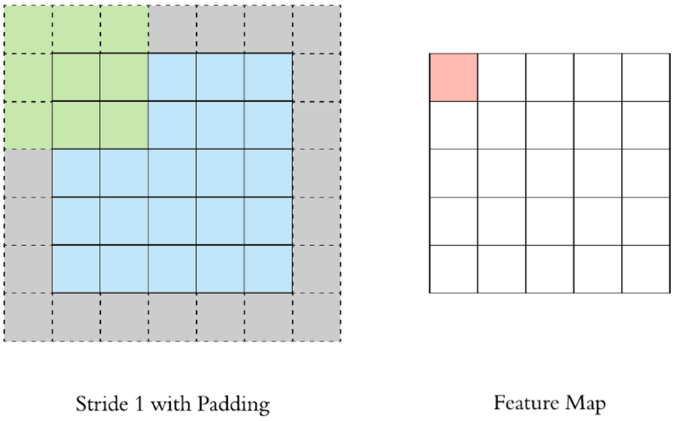
3.3 Stride and Padding

Stride denotes how many steps we are moving in each steps in convolution.By default it is one.



Convolution with Stride 1

We can observe that the size of output is smaller that input. To maintain the dimension of output as in input , we use padding. Padding is a process of adding zeros to the input matrix symmetrically. In the following example,the extra grey blocks denote the padding. It is used to make the dimension of output same as input.



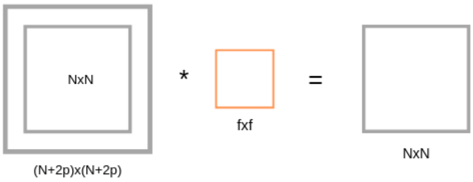
Stride 1 with Padding 1

Let say 'p' is the padding

Initially(without padding)

```
(N x N) * (F x F) = (N-F+1)x(N-F+1)---(1)
```

After applying padding



After applying padding

If we apply filter F x F in (N+2p) x (N+2p) input matrix with padding, then we will get output matrix dimension (N+2p-F+1) x (N+2p-F+1). As we know that after applying padding we will get the same dimension as original input dimension (N x N). Hence we have,

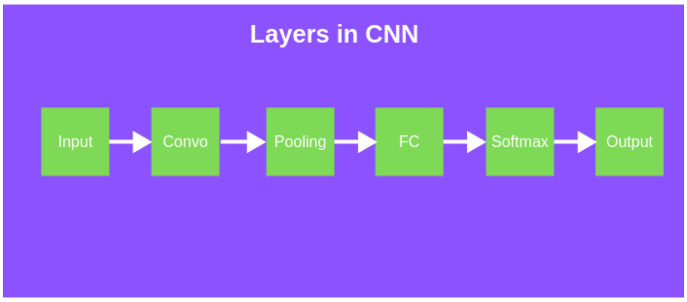
```
(N+2p-F+1)x(N+2p-F+1) equivalent to NxN
 N+2p-F+1 = N ---(2)
 p = (F-1)/2 ---(3)
```

The equation (3) clearly shows that Padding depends on the dimension of filter.

4. Layers in CNN

There are five different layers in CNN

- Input layer
- Convo layer (Convo + ReLU)
- Pooling layer
- Fully connected(FC) layer
- Softmax/logistic layer
- Output layer



Different layers of CNN

4.1 Input Layer

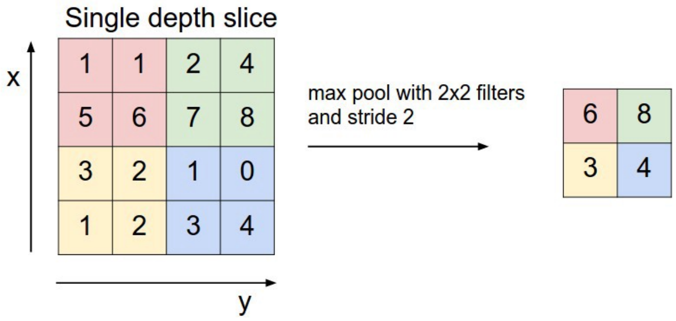
Input layer in CNN should contain image data. Image data is represented by three dimensional matrix as we saw earlier. You need to reshape it into a single column. Suppose you have image of dimension 28 x 28 =784, you need to convert it into 784 x 1 before feeding into input. If you have “m” training examples then dimension of input will be (784, m).

4.2. Convo Layer

Convo layer is sometimes called feature extractor layer because features of the image are get extracted within this layer. First of all, a part of image is connected to Convo layer to perform convolution operation as we saw earlier and calculating the dot product between receptive field(it is a local region of the input image that has the same size as that of filter) and the filter. Result of the operation is single integer of the output volume. Then we slide the filter over the next receptive field of the same input image by a Stride and do the same operation again. We will repeat the same process again and again until we go through the whole image. The output will be the input for the next layer.

Convo layer also contains ReLU activation to make all negative value to zero.

4.3. Pooling Layer



1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

6	8
3	4

Source : CS231n Convolutional Neural Network

Pooling layer is used to reduce the spatial volume of input image after convolution. It is used between two convolution layer. If we apply FC after Convo layer without applying pooling or max pooling, then it will be computationally expensive and we don't want it. So, the max pooling is only way to reduce the spatial volume of input image. In the above example, we have applied max pooling in single depth slice with Stride of 2. You can observe the 4 x 4 dimension input is reduce to 2 x 2 dimension.

There is no parameter in pooling layer but it has two hyperparameters — Filter(F) and Stride(S).

In general, if we have input dimension W1 x H1 x D1, then

$W2 = (W1-F)/S+1$

$H2 = (H1-F)/S+1$

$D2 = D1$

Where W2, H2 and D2 are the width, height and depth of output.

4.4. Fully Connected Layer(FC)

Fully connected layer involves weights, biases, and neurons. It connects neurons in one layer to neurons in another layer. It is used to classify images between different category by training.

4.5. Softmax / Logistic Layer

Softmax or Logistic layer is the last layer of CNN. It resides at the end of FC layer. Logistic is used for binary classification and softmax is for multi-classification.

4.6. Output Layer

Output layer contains the label which is in the form of one-hot encoded.

Now you have a good understanding of CNN. Let's implement a CNN in Keras.

5. Keras Implementation

We will use CIFAR-10 dataset to build a CNN image classifier. CIFAR-10 dataset has 10 different labels

- Airplane
- Automobile

- Bird
- Cat
- Deer
- Dog
- Frog
- Horse
- Ship
- Truck

It has 50,000 training data and 10,000 testing image data. Image size in CIFAR-10 is 32 x 32 x 3. It comes with Keras library.

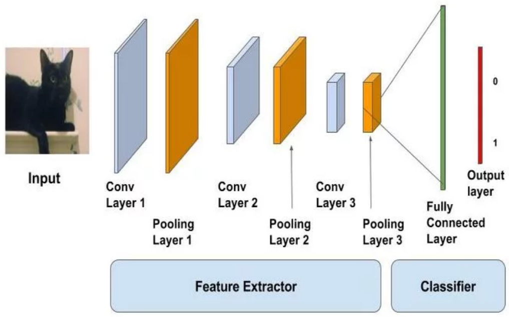


Fig. Model visualization

If you are using google colaboratory, then make sure you are using GPU. To check whether your GPU is on or not. Try following code.

```
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

cifar-10.py hosted with ♡ by GitHub view raw

Output:

```
Found GPU at: /device:GPU:0
```

First of all, import all necessary modules and libraries.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from __future__ import print_function
import keras
from keras.models import Sequential
from keras.utils import to_categorical
from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout, Flatten
```

cifar-10.py hosted with ♡ by GitHub view raw

Then load the dataset and split it into train and test sets.

```
from keras.datasets import cifar10
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()
```

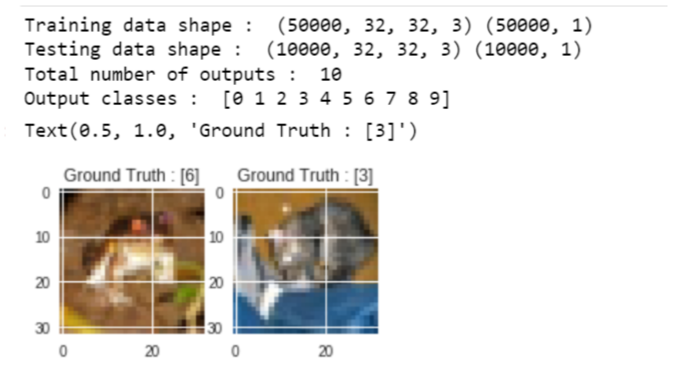
cifar-10.py hosted with ♡ by GitHub view raw

We will print training sample shape, test sample shape and total number of classes present in CIFAR-10. There are 10 classes as we saw earlier. For the sake of example, we will print two example image from training set and test set.

```python
print('Training data shape : ', train_images.shape, train_labels.shape)

print('Testing data shape : ', test_images.shape, test_labels.shape)

# Find the unique numbers from the train labels
classes = np.unique(train_labels)
nClasses = len(classes)
print('Total number of outputs : ', nClasses)
print('Output classes : ', classes)

plt.figure(figsize=[4,2])

# Display the first image in training data
plt.subplot(121)
plt.imshow(train_images[0,:,:], cmap='gray')
plt.title("Ground Truth : {}".format(train_labels[0]))

# Display the first image in testing data
plt.subplot(122)
plt.imshow(test_images[0,:,:], cmap='gray')
plt.title("Ground Truth : {}".format(test_labels[0]))
```

cifar-10.py hosted with ♡ by GitHub view raw

Output:



```
Training data shape :  (50000, 32, 32, 3) (50000, 1)
Testing data shape :  (10000, 32, 32, 3) (10000, 1)
Total number of outputs :  10
Output classes :  [0 1 2 3 4 5 6 7 8 9]
Text(0.5, 1.0, 'Ground Truth : [3]')
```

Find the shape of input image then reshape it into input format for training and testing sets. After that change all datatypes into floats.

```python
nRows,nCols,nDims = train_images.shape[1:]
train_data = train_images.reshape(train_images.shape[0], nRows, nCols, nDims)
test_data = test_images.reshape(test_images.shape[0], nRows, nCols, nDims)
input_shape = (nRows, nCols, nDims)

train_data = train_data.astype('float32')
test_data = test_data.astype('float32')
```

cifar-10.py hosted with ♡ by GitHub view raw

Normalize the data between 0–1 by dividing train data and test data with 255 then convert all labels into one-hot vector with *to_catagorical()* function.

```python

train_data /= 255
test_data /= 255

train_labels_one_hot = to_categorical(train_labels)
test_labels_one_hot = to_categorical(test_labels)
```

cifar-10.py hosted with ♡ by GitHub view raw

Display the change for category label using one-hot encoding.

```python
print('Original label 0 : ', train_labels[0])
print('After conversion to categorical ( one-hot ) : ', train_labels_one_hot[0])
```

cifar-10.py hosted with ♡ by GitHub view raw

Output:

```
Original label 0 :  [6]
After conversion to categorical ( one-hot ) :
[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

Now create our model. We will add up Convo layers followed by pooling layers. Then we will connect Dense(FC) layer to predict the classes. Input data fed to first Convo layer, output of that Convo layer acts as input for next Convo layer and so on. Finally data is fed to FC layer which try to predict the correct labels.

```python
def createModel():
    model = Sequential()
    # The first two layers with 32 filters of window size 3x3
    model.add(Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=input_shape))
    model.add(Conv2D(32, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Flatten())
    model.add(Dense(512, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(nClasses, activation='softmax'))

    return model
```

cifar-10.py hosted with ♡ by GitHub view raw

Initialize all parameters and compile our model with rmsprops optimizer. There are many optimizers for example adam, SGD, GradientDescent, Adagrad, Adadelta and Adamax ,feel free to experiment with it. Here batch is 256 with 50 epochs.

```python
model1 = createModel()
batch_size = 256
epochs = 50
model1.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

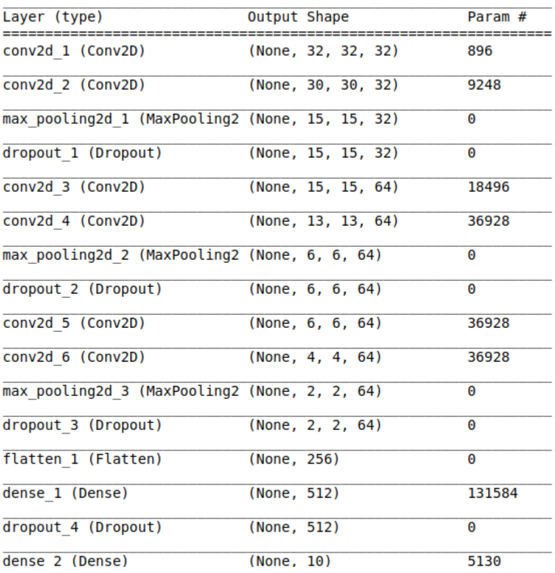
cifar-10.py hosted with ♡ by GitHub view raw

model.summary() is used to see all parameters and shapes in each layers in our models. You can observe that total parameters are 276, 138 and total trainable parameters are 276, 138. Non-trainable parameter is 0.

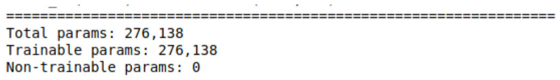
```python
model1.summary()
```

cifar-10.py hosted with ♡ by GitHub view raw

Output:



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        896

conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248

max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0

dropout_1 (Dropout)          (None, 15, 15, 32)        0

conv2d_3 (Conv2D)            (None, 15, 15, 64)        18496

conv2d_4 (Conv2D)            (None, 13, 13, 64)        36928

max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0

dropout_2 (Dropout)          (None, 6, 6, 64)          0

conv2d_5 (Conv2D)            (None, 6, 6, 64)          36928

conv2d_6 (Conv2D)            (None, 4, 4, 64)          36928

max_pooling2d_3 (MaxPooling2 (None, 2, 2, 64)          0

dropout_3 (Dropout)          (None, 2, 2, 64)          0

flatten_1 (Flatten)          (None, 256)               0

dense_1 (Dense)              (None, 512)               131584

dropout_4 (Dropout)          (None, 512)               0

dense_2 (Dense)              (None, 10)                5130
```



```
=================================================================
Total params: 276,138
Trainable params: 276,138
Non-trainable params: 0
```

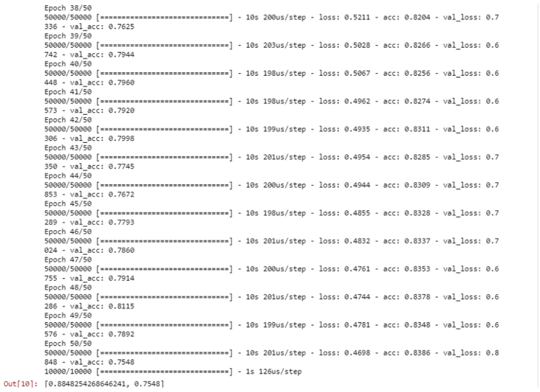
Model Summary

After compiling our model, we will train our model by *fit()* method, then evaluate it.

```
history = model1.fit(train_data, train_labels_one_hot, batch_size=batch_size, epochs=epochs, verb
                     validation_data=(test_data, test_labels_one_hot))
model1.evaluate(test_data, test_labels_one_hot)
```

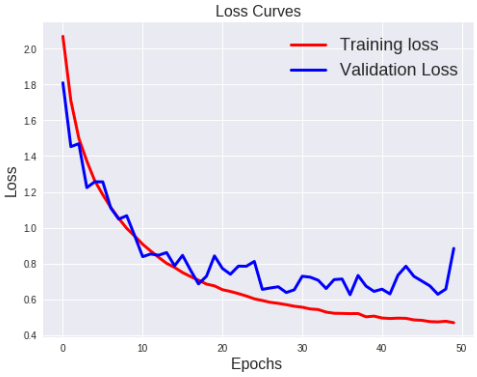
cifar-10.py hosted with ♡ by GitHub view raw

Output:

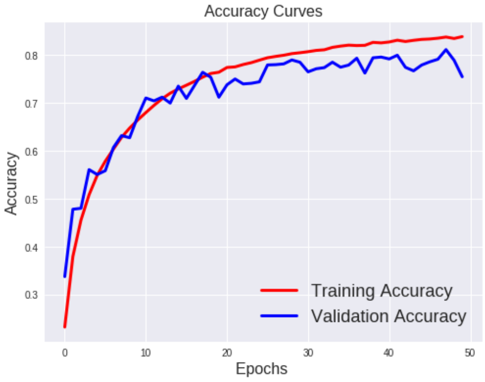


```
Epoch 38/50
50000/50000 [==============================] - 10s 200us/step - loss: 0.5211 - acc: 0.8204 - val_loss: 0.7
336 - val_acc: 0.7625
Epoch 39/50
50000/50000 [==============================] - 10s 203us/step - loss: 0.5028 - acc: 0.8266 - val_loss: 0.6
742 - val_acc: 0.7944
Epoch 40/50
50000/50000 [==============================] - 10s 198us/step - loss: 0.5067 - acc: 0.8256 - val_loss: 0.6
448 - val_acc: 0.7960
Epoch 41/50
50000/50000 [==============================] - 10s 198us/step - loss: 0.4962 - acc: 0.8274 - val_loss: 0.6
573 - val_acc: 0.7920
Epoch 42/50
50000/50000 [==============================] - 10s 199us/step - loss: 0.4935 - acc: 0.8311 - val_loss: 0.6
306 - val_acc: 0.7998
Epoch 43/50
50000/50000 [==============================] - 10s 201us/step - loss: 0.4954 - acc: 0.8285 - val_loss: 0.7
350 - val_acc: 0.7745
Epoch 44/50
50000/50000 [==============================] - 10s 200us/step - loss: 0.4944 - acc: 0.8309 - val_loss: 0.7
853 - val_acc: 0.7672
Epoch 45/50
50000/50000 [==============================] - 10s 198us/step - loss: 0.4855 - acc: 0.8328 - val_loss: 0.7
289 - val_acc: 0.7793
Epoch 46/50
50000/50000 [==============================] - 10s 201us/step - loss: 0.4832 - acc: 0.8337 - val_loss: 0.7
024 - val_acc: 0.7860
Epoch 47/50
50000/50000 [==============================] - 10s 200us/step - loss: 0.4761 - acc: 0.8353 - val_loss: 0.6
755 - val_acc: 0.7914
Epoch 48/50
50000/50000 [==============================] - 10s 201us/step - loss: 0.4744 - acc: 0.8378 - val_loss: 0.6
286 - val_acc: 0.8115
Epoch 49/50
50000/50000 [==============================] - 10s 199us/step - loss: 0.4781 - acc: 0.8348 - val_loss: 0.6
576 - val_acc: 0.7892
Epoch 50/50
50000/50000 [==============================] - 10s 201us/step - loss: 0.4698 - acc: 0.8386 - val_loss: 0.8
848 - val_acc: 0.7548
10000/10000 [==============================] - 1s 126us/step
Out[10]: [0.8848254268646241, 0.7548]
```

After training we got 83.86% accuracy and 75.48% validation accuracy. It is not actually bad at all.



Loss vs Epochs graph



Accuracy vs Epochs graph

Conclusion:

Congratulations! You have made Convolutional Neural Network in Keras by understanding its basic concepts. Feel free to experiment with it by changing its hyperparameters and let me know in the comment section.

You can find all code in my Github

References:

1. CS231n Convolutional Neural Network
2. Keras documentation
3. Deep Learning Lab
4. deeplearning.ai course on Convolutional Nneural Netowrks

Machine Learning Deep Learning Computer Vision Convolutional Neural Net Neural Networks