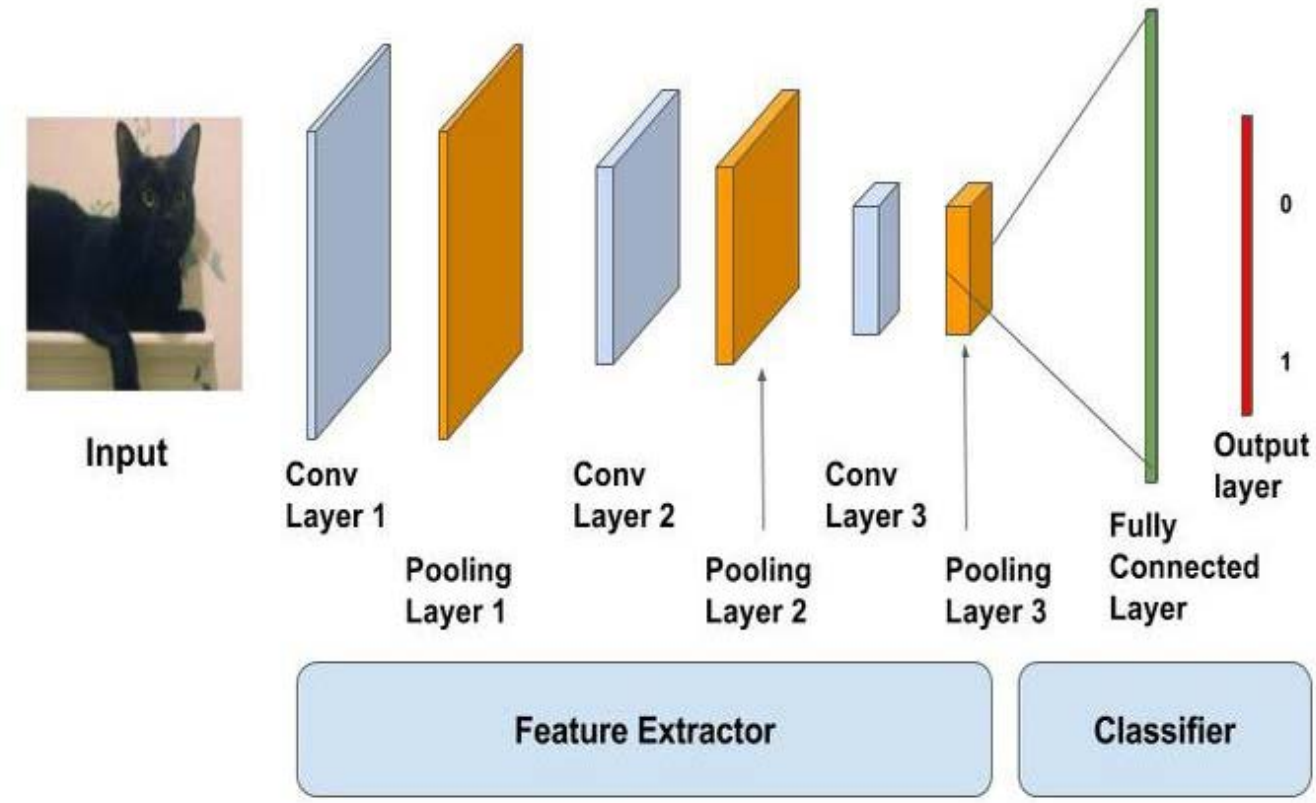
Image Classification using Convolutional Neural Networks in Keras

NOVEMBER 29, 2017 BY VIKAS GUPTA — 5 COMMENTS



In this tutorial, we will learn the basics of Convolutional Neural Networks (CNNs) and how to use them for an Image Classification task. We will also see how data augmentation helps in improving the performance of the network. We discussed Feedforward Neural Networks, Activation Functions, and Basics of Keras in the previous tutorials. We will use the MNIST and CIFAR10 datasets for illustrating various concepts.

This post is part of the series on Deep Learning for Beginners, which consists of the following tutorials :

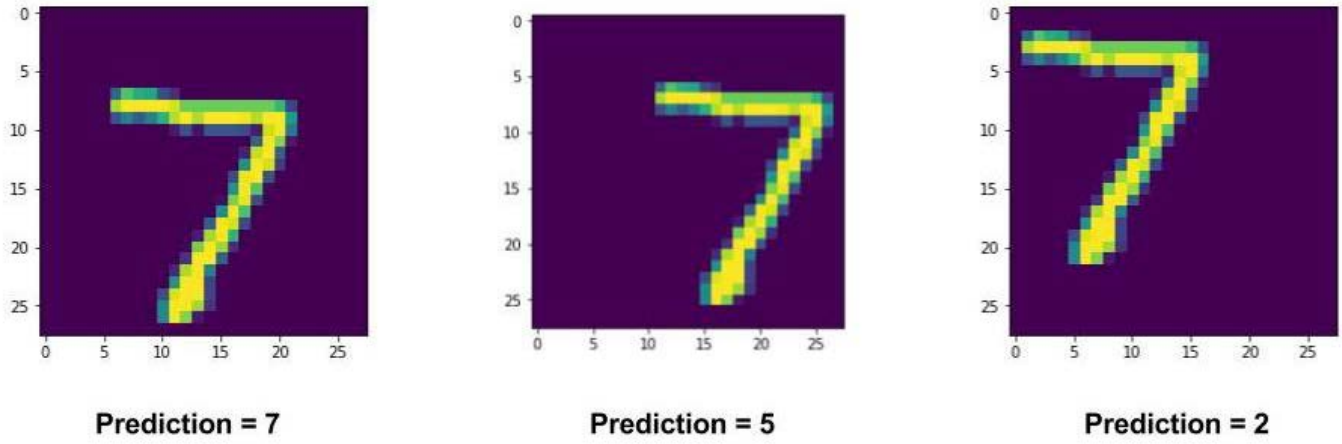
1. Neural Networks : A 30,000 Feet View for Beginners
2. Installation of Deep Learning frameworks (Tensorflow and Keras with CUDA support)
3. Introduction to Keras
4. Understanding Feedforward Neural Networks
5. Image Classification using Feedforward Neural Networks
6. Image Recognition using Convolutional Neural Network
7. Understanding Activation Functions
8. Understanding AutoEncoders using Tensorflow
9. Image Classification using pre-trained models in Keras
10. Transfer Learning using pre-trained models in Keras
11. Fine-tuning pre-trained models in Keras
12. More to come . . .

1. Motivation

In our previous article on Image Classification, we used a Multilayer Perceptron on the MNIST digits dataset. The performance was pretty good as we achieved 98.3% accuracy on test data. But there was a problem with that approach. In our training dataset, all images are centered. If the images in the test set are off-center, then the MLP approach fails miserably. We want the network to be Translation-Invariant.

Given below is an example of the number 7 being pushed to the top-left and bottom-right. The classifier predicts it correctly for the centered image but fails in the other two cases. To make it work for these images, either we have to train separate MLPs for different locations or we have to make sure that we

have all these variations in the training set as well, which I would say is difficult, if not impossible.



The Fully connected network tries to learn global features or patterns. It acts as a good classifier.

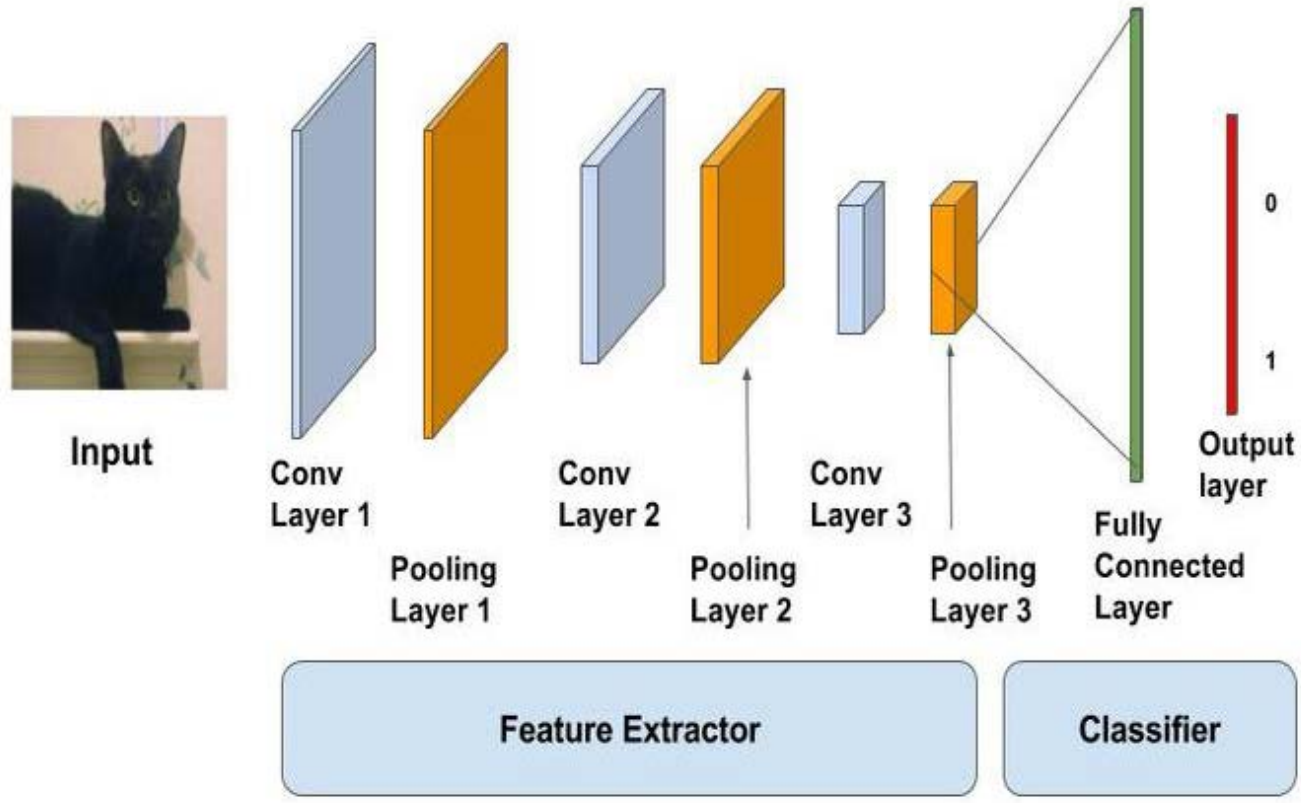
Another major problem with a fully connected classifier is that the number of parameters increases very fast since each node in layer L is connected to a node in layer L-1. So it is not feasible to design very deep networks using an MLP structure alone.

Both the above problems are solved to a great extent by using Convolutional Neural Networks which we will see in the next section. We will first describe the concepts involved in a Convolutional Neural Network in brief and then see an implementation of CNN in Keras so that you get a hands-on experience.

2. Convolutional Neural Network

Convolutional Neural Networks are a form of Feedforward Neural Networks. Given below is a schema of a typical CNN. The first part consists of Convolutional and max-pooling layers which act as the feature extractor. The

second part consists of the fully connected layer which performs non-linear transformations of the extracted features and acts as the classifier.



In the above diagram, the input is fed to the network of stacked Conv, Pool and Dense layers. The output can be a softmax layer indicating whether there is a cat or something else. You can also have a sigmoid layer to give you a probability of the image being a cat. Let us see the two layers in detail.

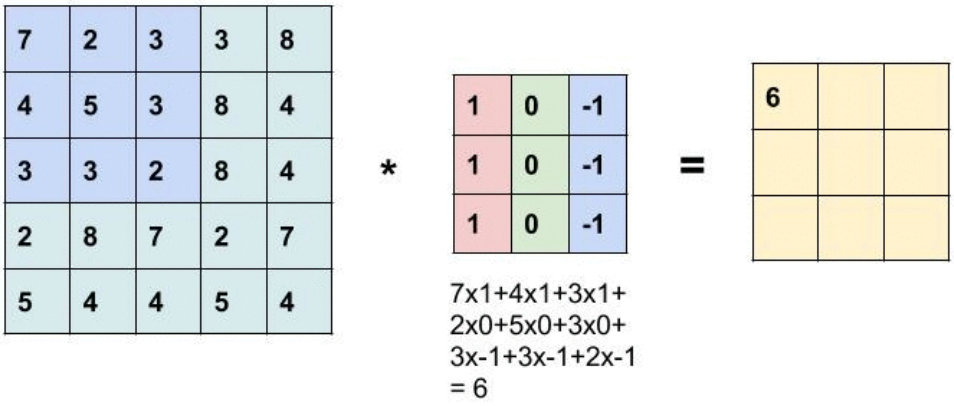
2.1. Convolutional Layer

The convolutional layer can be thought of as the eyes of the CNN. The neurons in this layer look for specific features. If they find the features they are looking for, they produce a high activation.

Convolution can be thought of as a weighted sum between two signals (in terms of signal processing jargon) or functions (in terms of mathematics). In

image processing, to calculate convolution at a particular location (x, y), we extract k x k sized chunk from the image centered at location (x, y). We then multiply the values in this chunk element-by-element with the convolution filter (also sized k x k) and then add them all to obtain a single output. That's it! Note that k is termed as the kernel size.

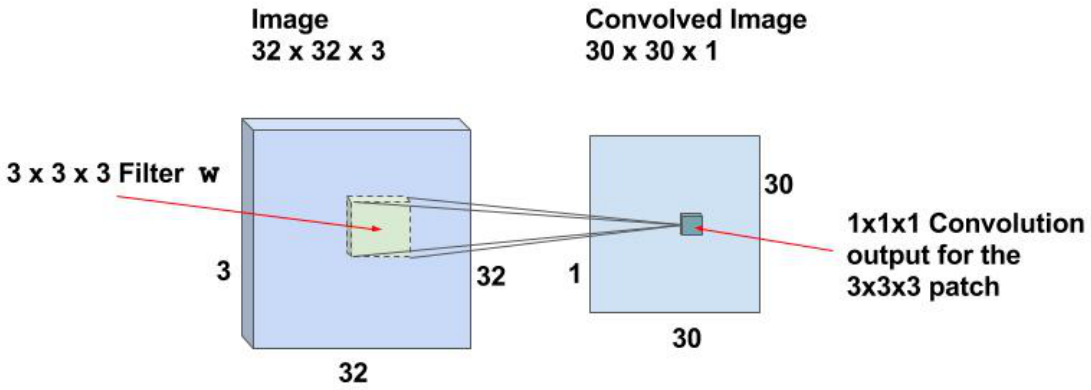
An example of convolution operation on a matrix of size 5×5 with a kernel of size 3×3 is shown below :



The convolution kernel is slid over the entire matrix to obtain an activation map.

Let's look at a concrete example and understand the terms. Suppose, the input image is of size 32x32x3. This is nothing but a 3D array of depth 3. Any convolution filter we define at this layer must have a depth equal to the depth of the input. So we can choose convolution filters of depth 3 (e.g. 3x3x3 or 5x5x3 or 7x7x3 etc.). Let's pick a convolution filter of size 3x3x3. So, referring

to the above example, here the convolutional kernel will be a cube instead of a square.



If we can perform the convolution operation by sliding the 3x3x3 filter over the entire 32x32x3 sized image, we will obtain an output image of size 30x30x1. This is because the convolution operation is not defined for a strip 2 pixels wide around the image. We have to ensure the filter is always inside the image. So 1 pixel is stripped away from left, right, top and bottom of the image.

> *The same filters are slid over the entire image to find the relevant features. This makes the CNNs Translation Invariant.*

2.1.1. Activation Maps

For a 32x32x3 input image and filter size of 3x3x3, we have 30x30x1 locations and there is a neuron corresponding to each location. Then 30x30x1

outputs or activations of all neurons are called the activation maps. The activation map of one layer serves as the input to the next layer.

2.1.2. Shared weights and biases

In our example, there are 30×30 = 900 neurons because there are that many locations where the 3x3x3 filter can be applied. Unlike traditional neural nets where weights and biases of neurons are independent of each other, in case of CNNs the neurons corresponding to one filter in a layer share the same weights and biases.

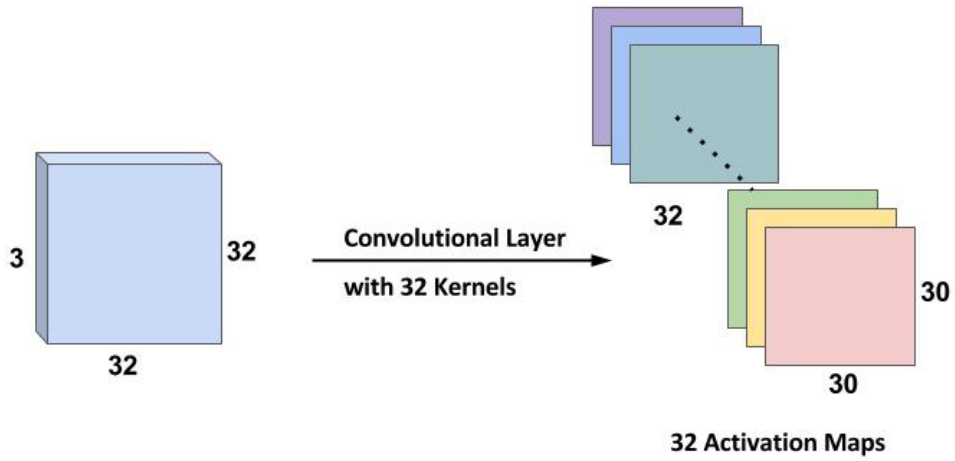
2.1.3. Stride

In the above case, we slid the window by 1 pixel at a time. We can also slide the window by more than 1 pixel. This number is called the stride.

2.1.4. Multiple Filters

Typically, we use more than 1 filter in one convolution layer. If we use 32 filters we will have an activation map of size 30x30x32. Please refer to Figure below for a graphical view.
Note that all neurons associated with the same filter share the same weights and biases. So the number of weights while using 32 filters is simply 3x3x3x32 = 288 and the number of biases is 32.

The 32 Activation maps obtained from applying the convoltional Kernels is shown below.



2.1.5. Zero padding

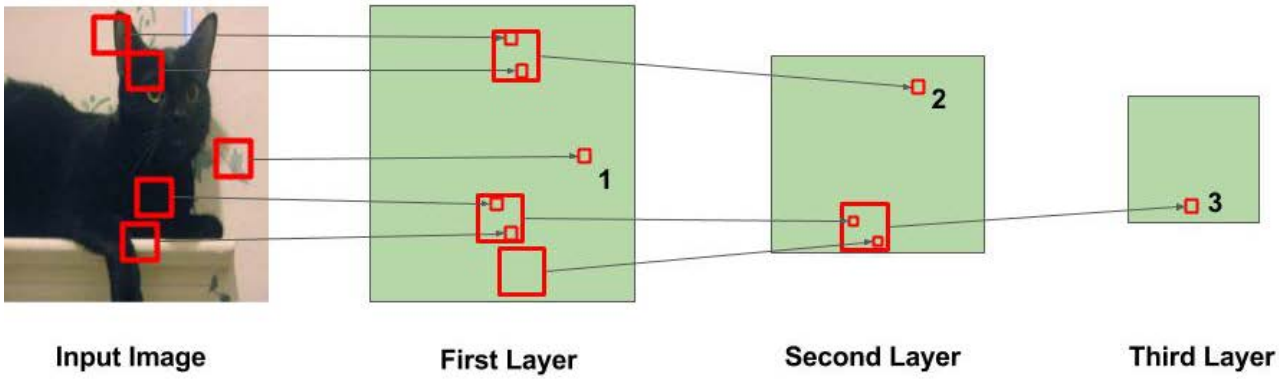
As you can see, after each convolution, the output reduces in size (as in this case we are going from 32×32 to 30×30). For convenience, it's a standard practice to pad zeros to the boundary of the input layer such that the output is the same size as input layer. So, in this example, if we add a padding of size 1 on both sides of the input layer, the size of the output layer will be 32x32x32 which makes implementation simpler as well. Let's say you have an input of size NxN, a filter of size F and you are using stride S and a zero padding of size P is added to the input image. Then, the output will be of size MxM where,

$$M = \frac{N-F+2P}{S} + 1$$

We can calculate the padding required so that the input and the output dimensions are the same by setting in the above equation and solving for P.

2.2. CNNs learn Hierarchical features

Let’s discuss how CNNs learn hierarchical features.



In the above figure, the big squares indicate the region over which the convolution operation is performed and the small squares indicate the output of the operation which is just a number. The following observations are to be noted :

- In the first layer, the square marked 1 is obtained from the area in the image where the leaves are painted.
- In the second layer, the square marked 2 is obtained from the bigger square in Layer 1. The numbers in this square are obtained from multiple regions from the input image. Specifically, the whole area around the left ear of the cat is responsible for the value at the square marked 2.
- Similarly, in the third layer, this cascading effect results in the square marked 3 being obtained from a large region around the leg area.

We can say from the above that the initial layers are looking at smaller regions of the image and thus can only learn simple features like edges / corners etc. As we go deeper into the network, the neurons get information from larger parts of the image and from various other neurons. Thus, the neurons at the later layers can learn more complicated features like eyes / legs and what not!

2.3. Max Pooling Layer

Pooling layer is mostly used immediately after the convolutional layer to reduce the spatial size (only width and height, not depth). This reduces the number of parameters, hence computation is reduced. Using fewer parameters avoids overfitting.

Note: Overfitting is the condition when a trained model works very well on training data, but does not work very well on test data.

The most common form of pooling is Max pooling where we take a filter of size p and apply the maximum operation over the sized part of the image.

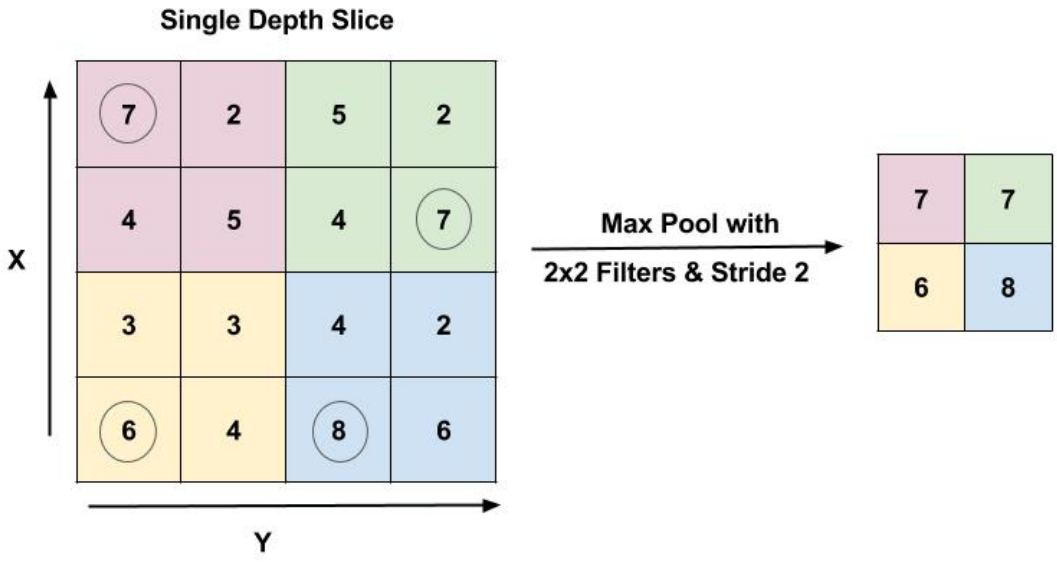


Figure : Max pool layer with filter size 2×2 and stride 2 is shown. The output is the max value in a 2×2 region shown using encircled digits.

The most common pooling operation is done with the filter of size 2×2 with a stride of 2. It essentially reduces the size of input by half.

Now let's take a break from the theoretical discussion and jump into the implementation of a CNN.

3. Implementing CNNs in Keras

Download Code

To easily follow along this tutorial, please download code by clicking on the button below. It's FREE!

DOWNLOAD CODE

3.1. The Dataset – CIFAR10

The CIFAR10 dataset comes bundled with Keras. It has 50,000 training images and 10,000 test images. There are 10 classes like airplanes, automobiles, birds, cats, deer, dog, frog, horse, ship and truck. The images are of size 32×32. Given below are a few examples.

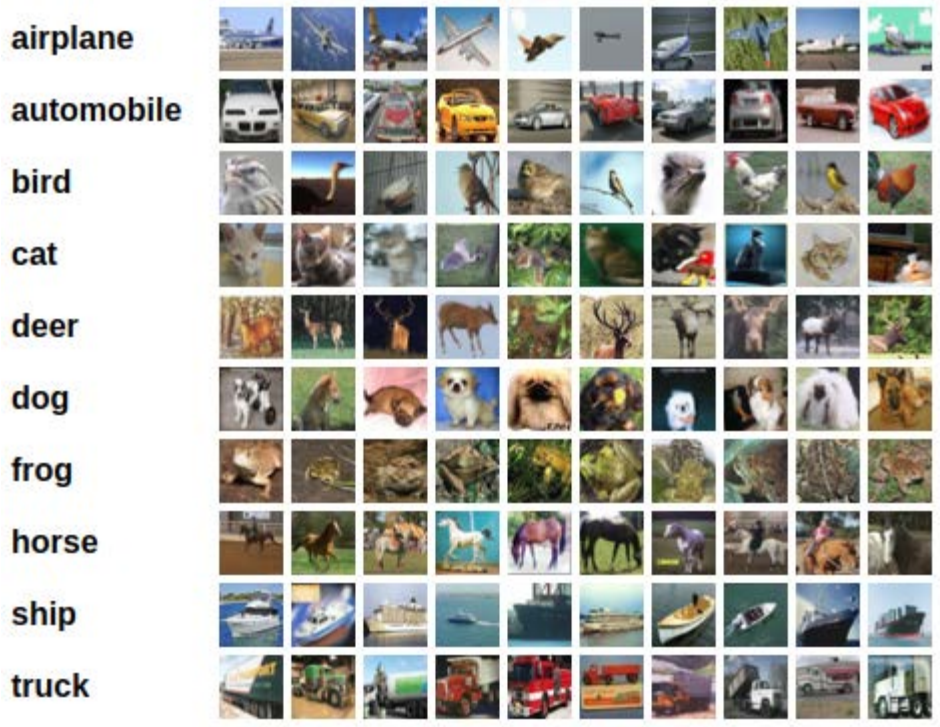


Image Credit : Alex Krizhevsky

3.2. The Network

For implementing a CNN, we will stack up Convolutional Layers, followed by Max Pooling layers. We will also include Dropout to avoid overfitting. Finally, we will add a fully connected (Dense) layer followed by a softmax layer. Given below is the model structure.

```
1  from keras.models import Sequential
2  from keras.layers import Dense, Conv2D, MaxPooling2D, Dropout, Flatten
3
4  def createModel():
5      model = Sequential()
6      model.add(Conv2D(32, (3, 3), padding='same', activation='relu',
7  input_shape=input_shape))
8      model.add(Conv2D(32, (3, 3), activation='relu'))
9      model.add(MaxPooling2D(pool_size=(2, 2)))
10     model.add(Dropout(0.25))
11
12     model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
13     model.add(Conv2D(64, (3, 3), activation='relu'))
14     model.add(MaxPooling2D(pool_size=(2, 2)))
15     model.add(Dropout(0.25))
16
17     model.add(Conv2D(64, (3, 3), padding='same', activation='relu'))
18     model.add(Conv2D(64, (3, 3), activation='relu'))
19     model.add(MaxPooling2D(pool_size=(2, 2)))
20     model.add(Dropout(0.25))
21
22     model.add(Flatten())
23     model.add(Dense(512, activation='relu'))
24     model.add(Dropout(0.5))
25     model.add(Dense(nClasses, activation='softmax'))
26
       return model
```

In the above code, we use 6 convolutional layers and 1 fully-connected layer. Line 6 and 7 adds convolutional layers with 32 filters / kernels with a window size of 3×3. Similarly, in line 10, we add a conv layer with 64 filters. In line 8, we add a max pooling layer with window size 2×2. In line 9, we add a dropout layer with a dropout ratio of 0.25. In the final lines, we add the dense layer which performs the classification among 10 classes using a softmax layer.

If we check the model summary we can see the shapes of each layer.

```
_________________________________________________________________
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
```

It shows that since we have used padding in the first layer, the output shape is same as the input (32×32). But the second conv layer shrinks by 2 pixels in both dimensions. Also, the output size after pooling layer decreases by half since we have used a stride of 2 and a window size of 2×2. The final droupout layer has an output of 2x2x64. This has to be converted to a single array. This is done by the flatten layer which converts the 3D array into a 1D array of size 2x2x64 = 256. The final layer has 10 nodes since there are 10 classes.

3.3. Training the network

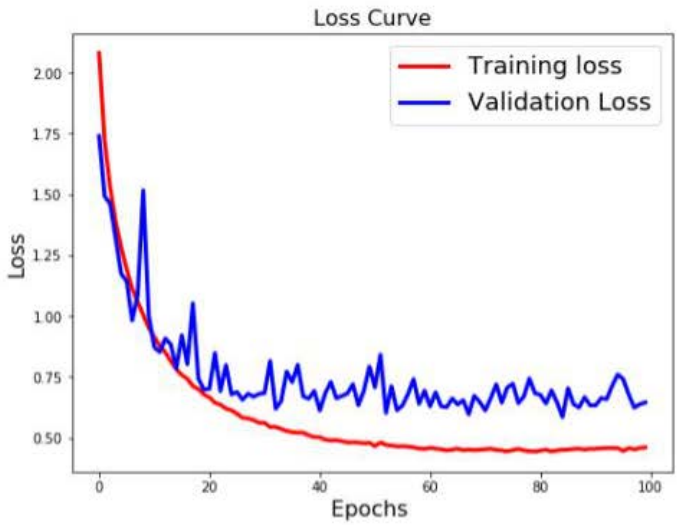
For training the network, we will follow the simple workflow of create -> compile -> fit described here. Since it is a 10 class classification problem, we will use a categorical cross entropy loss and use RMSProp optimizer to train the network. We will run it for some number of epochs. Here we run it for 100 epochs.

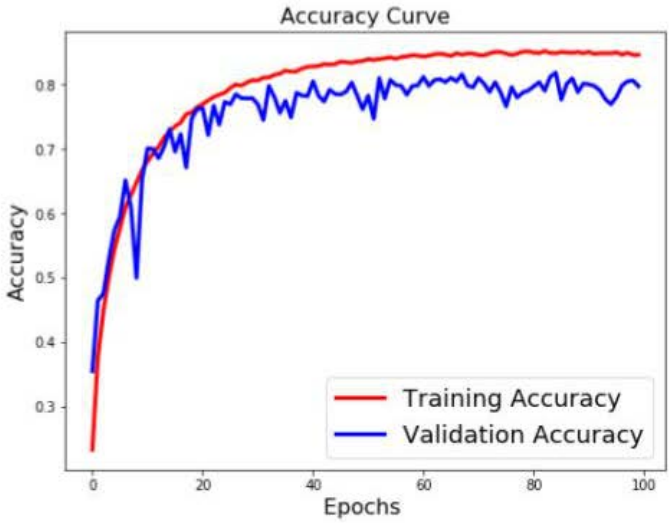
```
model1 = createModel()
batch_size = 256
epochs = 100
model1.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])

history = model1.fit(train_data, train_labels_one_hot, batch_size=batch_size,
epochs=epochs, verbose=1,
                  validation_data=(test_data, test_labels_one_hot))

model1.evaluate(test_data, test_labels_one_hot)
```

3.4. Loss & Accuracy Curves

Given below are the loss and accuracy curves.

```
# Loss Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['loss'],'r',linewidth=3.0)
plt.plot(history.history['val_loss'],'b',linewidth=3.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)

# Accuracy Curves
plt.figure(figsize=[8,6])
plt.plot(history.history['acc'],'r',linewidth=3.0)
plt.plot(history.history['val_acc'],'b',linewidth=3.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```





From the above curves, we can see that there is a considerable difference between the training and validation loss. This indicates that the network has tried to memorize the training data and thus, is able to get better accuracy on it. This is a sign of Overfitting. But we have already used Dropout in the network, then why is it still overfitting. Let us see if we can further reduce overfitting using something else.

4. Using Data Augmentation

One of the major reasons for overfitting is that you don't have enough data to train your network. Apart from regularization, another very effective way to counter Overfitting is Data Augmentation. It is the process of artificially creating more images from the images you already have by changing the size, orientation etc of the image. It can be a tedious task but fortunately, this can be done in Keras using the ImageDataGenerator instance.

```
1 from keras.preprocessing.image import ImageDataGenerator
2
```

```
ImageDataGenerator(
    rotation_range=10.,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.,
    zoom_range=.1.,
    horizontal_flip=True,
    vertical_flip=True)
```

In the above code, we have provided some of the operations that can be done using the ImageDataGenerator for data augmentation. This includes rotation of the image, shifting the image left/right/top/bottom by some amount, flip the image horizontally or vertically, shear or zoom the image etc. For the complete list, check the documentation. Some generated images are shown below.

4.1. Training with Data Augmentation

Similar to the previous section, we will create the model, but use data augmentation while training. We will use ImageDataGenerator for creating a generator which will feed the network.

```
from keras.preprocessing.image import ImageDataGenerator

model2 = createModel()
```

```
model2.compile(optimizer='rmsprop', loss='categorical_crossentropy',
metrics=['accuracy'])

batch_size = 256
epochs = 100
datagen = ImageDataGenerator(
#         zoom_range=0.2, # randomly zoom into images
#         rotation_range=10,  # randomly rotate images in the range (degrees, 0
to 180)
        width_shift_range=0.1,  # randomly shift images horizontally (fraction
of total width)
        height_shift_range=0.1,  # randomly shift images vertically (fraction of
total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

# Fit the model on the batches generated by datagen.flow().
history2 = model2.fit_generator(datagen.flow(train_data, train_labels_one_hot,
batch_size=batch_size),
                                steps_per_epoch=int(np.ceil(train_data.shape[0] /
float(batch_size))),
                                epochs=epochs,
                                validation_data=(test_data, test_labels_one_hot),
                                workers=4)

model2.evaluate(test_data, test_labels_one_hot)
```

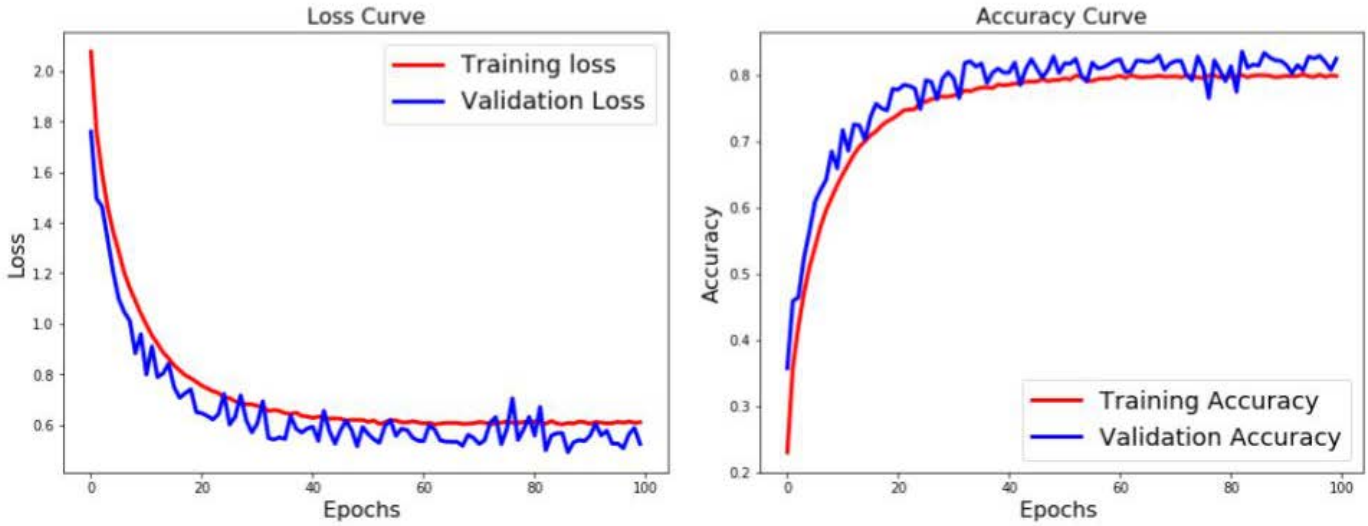
In the above code,

1. We first create the model and configure it.
2. Then we create an ImageDataGenerator object and configure it using parameters for horizontal flip, and image translation.
3. The datagen.flow() function generates batches of data, after performing the data transformations / augmentation specified during the instantiation of the data generator.
4. The fit_generator function will train the model using the data obtained in batches from the datagen.flow function.

4.2. Loss & Accuracy Curves

```
# Loss Curves
plt.figure(figsize=[8,6])
plt.plot(history2.history['loss'],'r',linewidth=3.0)
plt.plot(history2.history['val_loss'],'b',linewidth=3.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)

```

```
# Accuracy Curves
plt.figure(figsize=[8,6])
plt.plot(history2.history['acc'],'r',linewidth=3.0)
plt.plot(history2.history['val_acc'],'b',linewidth=3.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
```



The test accuracy is greater than training accuracy. This means that the model has generalized very well. This comes from the fact that the model has been trained on much worse data (for example – flipped images), so it is finding the normal test data easier to classify.

5. What next?

It looks like there were a lot of parameters to chose from and then training took a long time. We would not want to get tied down with these two problems when we are working on simple problems. Many researchers working in this field very generously open-source their trained models which have been trained on millions of images and for hundreds of hours on many GPUs. We

can leverage their models and try to use their trained models as the starting point rather than starting from scratch. We will learn how to do Transfer Learning and Fine-tuning in our next post.

References

https://github.com/fchollet/keras/blob/master/examples

Subscribe & Download Code

If you liked this article and would like to download code and example images used in this post, please subscribe to our newsletter. You will also receive a free Computer Vision Resource Guide. In our newsletter, we share OpenCV tutorials and examples written in C++/Python, and Computer Vision and Machine Learning algorithms and news.

Subscribe Now

FILED UNDER: DEEP LEARNING, IMAGE CLASSIFICATION, TUTORIAL TAGGED WITH: BEGINNERS, CONVOLUTIONAL NEURAL NETWORK, DEEP LEARNING, IMAGE CLASSIFICATION, KERAS

Search this website …

FREE COURSE FOR BEGINNERS

RESOURCES

Download Code (C++ / Python)

DISCLAIMER

This site is not affiliated with OpenCV.org

I am an entrepreneur with a love for Computer Vision and Machine Learning with a dozen years of experience (and a Ph.D.) in the field.

In 2007, right after finishing my Ph.D., I co-founded TAAZ Inc. with my advisor Dr. David Kriegman and Kevin Barnes. The scalability, and robustness of our computer vision and machine learning algorithms have

been put to rigorous test by more than 100M users who have tried our products. Read More…

RECENT POSTS

Image Alignment (Feature Based) using OpenCV (C++/Python)

Barcode and QR code Scanner using ZBar and OpenCV

Keras Tutorial : Fine-tuning using pre-trained models

OpenCV Transparent API

Face Reconstruction using EigenFaces (C++/Python)