2
Introduction to Convolutional Neural Networks

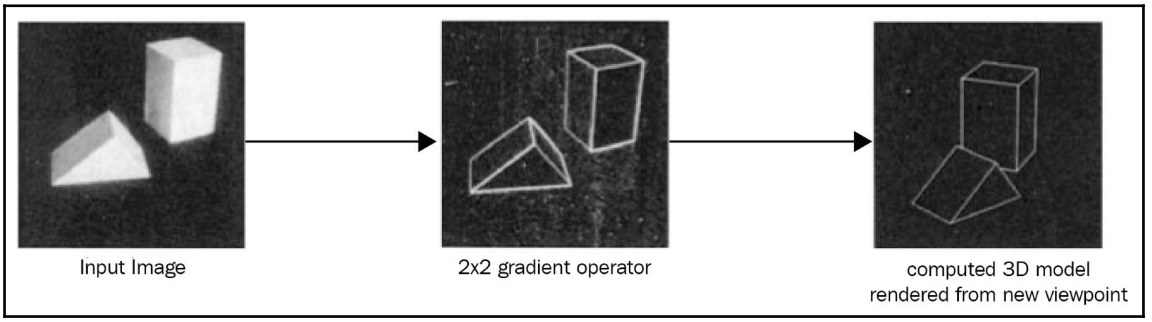
Convolutional Neural Networks (**CNNs**) are everywhere. In the last five years, we have seen a dramatic rise in the performance of visual recognition systems due to the introduction of deep architectures for feature learning and classification. CNNs have achieved good performance in a variety of areas, such as automatic speech understanding, computer vision, language translation, self-driving cars, and games such as Alpha Go. Thus, the applications of CNNs are almost limitless. DeepMind (from Google) recently published WaveNet, which uses a CNN to generate speech that mimics any human voice (`https://deepmind.com/blog/wavenet-generative-model-raw-audio/`).

In this chapter, we will cover the following topics:

- History of CNNs
- Overview of a CNN
- Image augmentation

History of CNNs

There have been numerous attempts to recognize pictures by machines for decades. It is a challenge to mimic the visual recognition system of the human brain in a computer. Human vision is the hardest to mimic and most complex sensory cognitive system of the brain. We will not discuss biological neurons here, that is, the primary visual cortex, but rather focus on artificial neurons. Objects in the physical world are three dimensional, whereas pictures of those objects are two dimensional. In this book, we will introduce neural networks without appealing to brain analogies. In 1963, computer scientist Larry Roberts, who is also known as the **father of computer vision**, described the possibility of extracting 3D geometrical information from 2D perspective views of blocks in his research dissertation titled **BLOCK WORLD**. This was the first breakthrough in the world of computer vision. Many researchers worldwide in machine learning and artificial intelligence followed this work and studied computer vision in the context of BLOCK WORLD. Human beings can recognize blocks regardless of any orientation or lighting changes that may happen. In this dissertation, he said that it is important to understand simple edge-like shapes in images. He extracted these edge-like shapes from blocks in order to make the computer understand that these two blocks are the same irrespective of orientation:

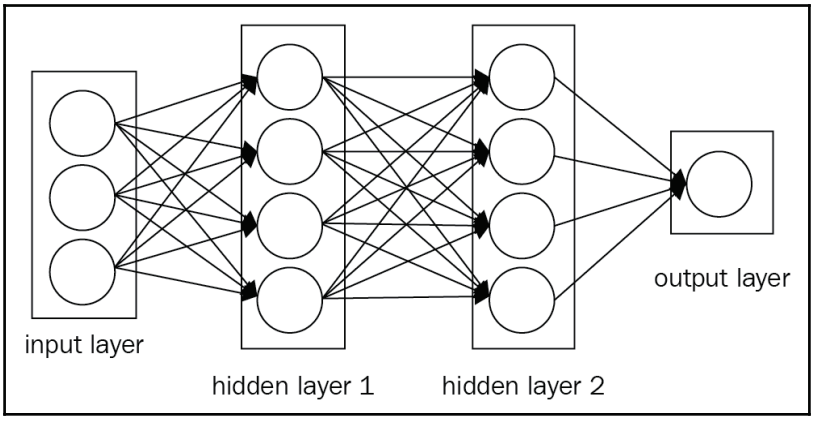


The vision starts with a simple structure. This is the beginning of computer vision as an engineering model. David Mark, an MIT computer vision scientist, gave us the next important concept, that vision is hierarchical. He wrote a very influential book named *VISION*. This is a simple book. He said that an image consists of several layers. These two principles form the basis of deep learning architecture, although they do not tell us what kind of mathematical model to use.

In the 1970s, the first visual recognition algorithm, known as the **generalized cylinder model**, came from the AI lab at Stanford University. The idea here is that the world is composed of simple shapes and any real-world object is a combination of these simple shapes. At the same time, another model, known as the **pictorial structure model**, was published from SRI Inc. The concept is still the same as the generalized cylinder model, but the parts are connected by springs; thus, it introduced a concept of variability. The first visual recognition algorithm was used in a digital camera by Fujifilm in 2006.

Convolutional neural networks

CNNs, or ConvNets, are quite similar to regular neural networks. They are still made up of neurons with weights that can be learned from data. Each neuron receives some inputs and performs a dot product. They still have a loss function on the last fully connected layer. They can still use a nonlinearity function. All of the tips and techniques that we learned from the last chapter are still valid for CNN. As we saw in the previous chapter, a regular neural network receives input data as a single vector and passes through a series of hidden layers. Every hidden layer consists of a set of neurons, wherein every neuron is fully connected to all the other neurons in the previous layer. Within a single layer, each neuron is completely independent and they do not share any connections. The last fully connected layer, also called the **output layer**, contains class scores in the case of an image classification problem. Generally, there are three main layers in a simple ConvNet. They are the **convolution layer**, the **pooling layer**, and the **fully connected layer**. We can see a simple neural network in the following image:



A regular three-layer neural network

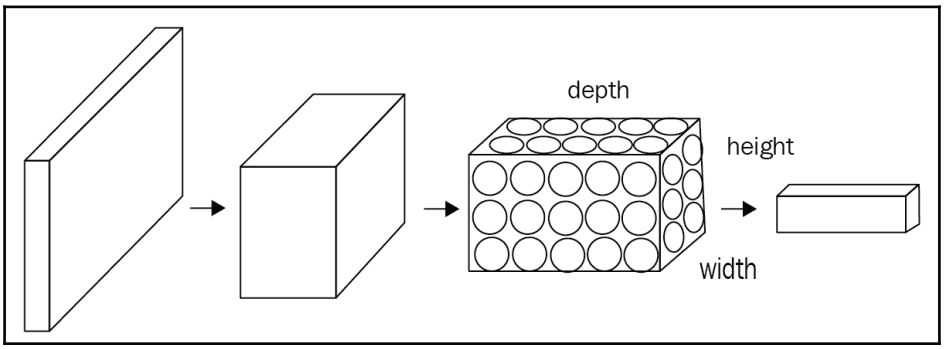
So, what changes? Since a CNN mostly takes images as input, this allows us to encode a few properties into the network, thus reducing the number of parameters.

In the case of real-world image data, CNNs perform better than **Multi-Layer Perceptrons** (**MLPs**). There are two reasons for this:

- In the last chapter, we saw that in order to feed an image to an MLP, we convert the input matrix into a simple numeric vector with no spatial structure. It has no knowledge that these numbers are spatially arranged. So, CNNs are built for this very reason; that is, to elucidate the patterns in multidimensional data. Unlike MLPs, CNNs understand the fact that image pixels that are closer in proximity to each other are more heavily related than pixels that are further apart:

  CNN = Input layer + hidden layer + fully connected layer

- CNNs differ from MLPs in the types of hidden layers that can be included in the model. A ConvNet arranges its neurons in three dimensions: **width**, **height**, and **depth**. Each layer transforms its 3D input volume into a 3D output volume of neurons using activation functions. For example, in the following figure, the red input layer holds the image. Thus its width and height are the dimensions of the image, and the depth is three since there are Red, Green, and Blue channels:



ConvNets are deep neural networks that share their parameters across space.

How do computers interpret images?

Essentially, every image can be represented as a matrix of pixel values. In other words, images can be thought of as a function (*f*) that maps from R^2 to *R*.

f(x, y) gives the intensity value at the position *(x, y)*. In practice, the value of the function ranges only from *0* to *255*. Similarly, a color image can be represented as a stack of three functions. We can write this as a vector of:

$$f(x, y) = [\, r(x,y)\ g(x,y)\ b(x,y)]$$

Or we can write this as a mapping:

$$f: R \times R \dashrightarrow R3$$

So, a color image is also a function, but in this case, a value at each *(x,y)* position is not a single number. Instead it is a vector that has three different light intensities corresponding to three color channels. The following is the code for seeing the details of an image as input to a computer.

Code for visualizing an image

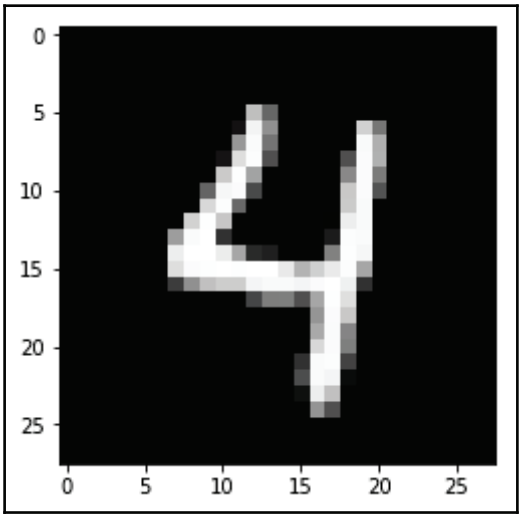
Let's take a look at how an image can be visualized with the following code:

```
#import all required lib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from skimage.io import imread
from skimage.transform import resize

# Load a color image in grayscale
image = imread('sample_digit.png',as_grey=True)
image = resize(image,(28,28),mode='reflect')
print('This image is: ',type(image),
         'with dimensions:', image.shape)

plt.imshow(image,cmap='gray')
```

We obtain the following image as a result:

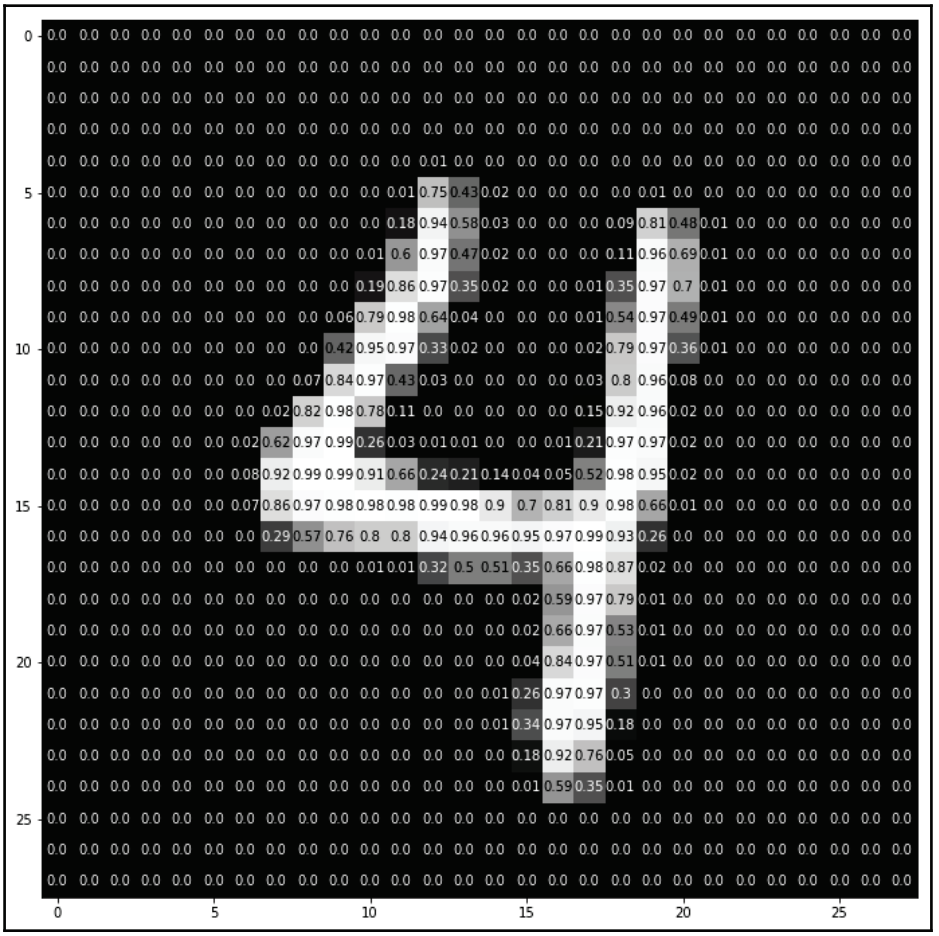



```
def visualize_input(img, ax):

    ax.imshow(img, cmap='gray')
    width, height = img.shape
    thresh = img.max()/2.5
    for x in range(width):
        for y in range(height):
            ax.annotate(str(round(img[x][y],2)), xy=(y,x),
                        horizontalalignment='center',
                        verticalalignment='center',
                        color='white' if img[x][y]<thresh else 'black')

fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
visualize_input(image, ax)
```

The following result is obtained:



In the previous chapter, we used an MLP-based approach to recognize images. There are two issues with that approach:

- It increases the number of parameters
- It only accepts vectors as input, that is, flattening a matrix to a vector

This means we must find a new way to process images, in which 2D information is not completely lost. CNNs address this issue. Furthermore, CNNs accept matrices as input. Convolutional layers preserve spatial structures. First, we define a convolution window, also called a **filter**, or **kernel**; then slide this over the image.

Dropout

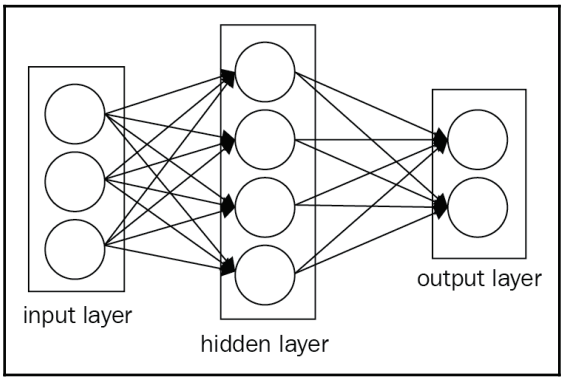
A neural network can be thought of as a search problem. Each node in the neural network is searching for correlation between the input data and the correct output data.

Dropout randomly turns nodes off while forward-propagating and thus helps ward off weights from converging to identical positions. After this is done, it turns on all the nodes and back-propagates. Similarly, we can set some of the layer's values to zero at random during forward propagation in order to perform dropout on a layer.

Use dropout only during training. Do not use it at runtime or on your testing dataset.

Input layer

The **input layer** holds the image data. In the following figure, the input layer consists of three inputs. In a **fully connected layer**, the neurons between two adjacent layers are fully connected pairwise but do not share any connection within a layer. In other words, the neurons in this layer have full connections to all activations in the previous layer. Therefore, their activations can be computed with a simple matrix multiplication, optionally adding a bias term. The difference between a fully connected and convolutional layer is that neurons in a convolutional layer are connected to a local region in the input, and that they also share parameters:



Convolutional layer

The main objective of convolution in relation to ConvNet is to extract features from the input image. This layer does most of the computation in a ConvNet. We will not go into the mathematical details of convolution here but will get an understanding of how it works over images.

The ReLU activation function is extremely useful in CNNs.

Convolutional layers in Keras

To create a convolutional layer in Keras, you must first import the required modules as follows:

```
from keras.layers import Conv2D
```

Then, you can create a convolutional layer by using the following format:

```
Conv2D(filters, kernel_size, strides, padding, activation='relu',
input_shape)
```

You must pass the following arguments:

- `filters`: The number of filters.
- `kernel_size`: A number specifying both the height and width of the (square) convolution window. There are also some additional optional arguments that you might like to tune.
- `strides`: The stride of the convolution. If you don't specify anything, this is set to one.
- `padding`: This is either `valid` or `same`. If you don't specify anything, the padding is set to `valid`.
- `activation`: This is typically `relu`. If you don't specify anything, no activation is applied. You are strongly encouraged to add a ReLU activation function to every convolutional layer in your networks.

It is possible to represent both `kernel_size` and `strides` as either a number or a tuple.

When using your convolutional layer as the first layer (appearing after the input layer) in a model, you must provide an additional `input_shape` argument—`input_shape`. It is a tuple specifying the height, width, and depth (in that order) of the input.

Please make sure that the `input_shape` argument is not included if the convolutional layer is not the first layer in your network.

There are many other tunable arguments that you can set to change the behavior of your convolutional layers:

- **Example 1**: In order to build a CNN with an input layer that accepts images of 200 x 200 pixels in grayscale. In such cases, the next layer would be a convolutional layer of 16 filters with width and height as 2. As we go ahead with the convolution we can set the filter to jump 2 pixels together. Therefore, we can build a convolutional, layer with a filter that doesn't pad the images with zeroes with the following code:

```
Conv2D(filters=16, kernel_size=2, strides=2, activation='relu',
input_shape=(200, 200, 1))
```

- **Example 2**: After we build our CNN model, we can have the next layer in it to be a convolutional layer. This layer will have 32 filters with width and height as 3, which would take the layer that was constructed in the previous example as its input. Here, as we proceed with the convolution, we will set the filter to jump one pixel at a time, such that the convolutional layer will be able to see all the regions of the previous layer too. Such a convolutional layer can be constructed with the help of the following code:

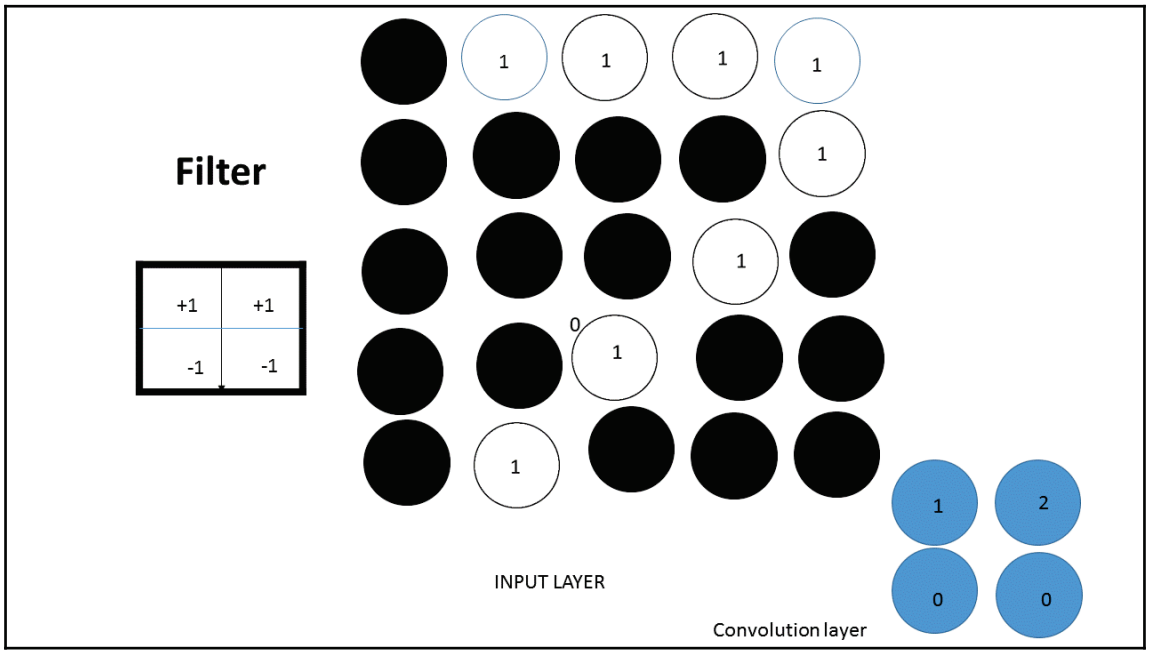
```
Conv2D(filters=32, kernel_size=3, padding='same',
activation='relu')
```

- **Example 3**: You can also construct convolutional layers in Keras of size 2 x 2, with 64 filters and a ReLU activation function. Here, the convolution utilizes a stride of 1 with padding set to `valid` and all other arguments set to their default values. Such a convolutional layer can be built using the following code:

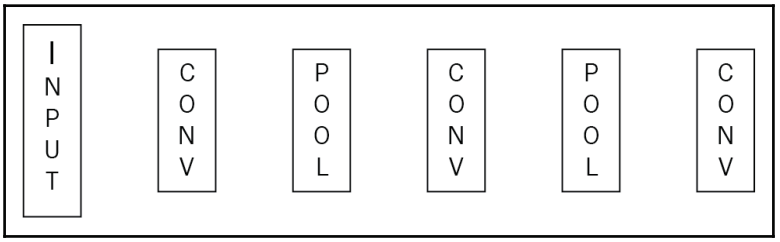
```
Conv2D(64, (2,2), activation='relu')
```

Pooling layer

As we have seen, a convolutional layer is a stack of feature maps, with one feature map for each filter. More filters increase the dimensionality of convolution. Higher dimensionality indicates more parameters. So, the pooling layer controls overfitting by progressively reducing the spatial size of the representation to reduce the number of parameters and computation. The pooling layer often takes the convolutional layer as input. The most commonly used pooling approach is **max pooling**. In addition to max pooling, pooling units can also perform other functions such as **average pooling**. In a CNN, we can control the behavior of the convolutional layer by specifying the size of each filter and the number of filters. To increase the number of nodes in a convolutional layer, we can increase the number of filters, and to increase the size of the pattern, we can increase the size of the filter. There are also a few other hyperparameters that can be tuned. One of them is the stride of the convolution. Stride is the amount by which the filter slides over the image. A stride of 1 moves the filter by 1 pixel horizontally and vertically. Here, the convolution becomes the same as the width and depth of the input image. A stride of 2 makes a convolutional layer of half of the width and height of the image. If the filter extends outside of the image, then we can either ignore these unknown values or replace them with zeros. This is known as **padding**. In Keras, we can set `padding = 'valid'` if it is acceptable to lose a few values. Otherwise, set `padding = 'same'`:



A very simple ConvNet looks like this:



Practical example – image classification

The convolutional layer helps to detect regional patterns in an image. The max pooling layer, present after the convolutional layer, helps reduce dimensionality. Here is an example of image classification using all the principles we studied in the previous sections. One important notion is to first make all the images into a standard size before doing anything else. The first convolution layer requires an additional `input.shape()` parameter. In this section, we will train a CNN to classify images from the CIFAR-10 database. CIFAR-10 is a dataset of 60,000 color images of 32 x 32 size. These images are labeled into 10 categories with 6,000 images each. These categories are airplane, automobile, bird, cat, dog, deer, frog, horse, ship, and truck. Let's see how to do this with the following code:

```
import keras
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))from keras.datasets import cifar10

# rescale [0,255] --> [0,1]
x_train = x_train.astype('float32')/255
from keras.utils import np_utils

# one-hot encode the labels
num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
# break training set into training and validation sets
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]

# print shape of training set
print('x_train shape:', x_train.shape)

# printing number of training, validation, and test images
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')x_test =
x_test.astype('float32')/255


from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same',
activation='relu',
                         input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.3))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(10, activation='softmax'))

model.summary()

# compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
                  metrics=['accuracy'])
from keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5',
verbose=1,
                               save_best_only=True)
```

```
hist = model.fit(x_train, y_train, batch_size=32, epochs=100,
          validation_data=(x_valid, y_valid), callbacks=[checkpointer],
          verbose=2, shuffle=True)
```

Image augmentation

While training a CNN model, we do not want the model to change any prediction based on the size, angle, and position of the image. The image is represented as a matrix of pixel values, so the size, angle, and position have a huge effect on the pixel values. To make the model more size-invariant, we can add different sizes of the image to the training set. Similarly, in order to make the model more rotation-invariant, we can add images with different angles. This process is known as **image data augmentation**. This also helps to avoid overfitting. Overfitting happens when a model is exposed to very few samples. Image data augmentation is one way to reduce overfitting, but it may not be enough because augmented images are still correlated. Keras provides an image augmentation class called `ImageDataGenerator` that defines the configuration for image data augmentation. This also provides other features such as:

- Sample-wise and feature-wise standardization
- Random rotation, shifts, shear, and zoom of the image
- Horizontal and vertical flip
- ZCA whitening
- Dimension reordering
- Saving the changes to disk

An augmented image generator object can be created as follows:

```
imagedatagen = ImageDataGenerator()
```

This API generates batches of tensor image data in real-time data augmentation, instead of processing an entire image dataset in memory. This API is designed to create augmented image data during the model fitting process. Thus, it reduces the memory overhead but adds some time cost for model training.

After it is created and configured, you must fit your data. This computes any statistics required to perform the transformations to image data. This is done by calling the `fit()` function on the data generator and passing it to the training dataset, as follows:

```
imagedatagen.fit(train_data)
```

The batch size can be configured, the data generator can be prepared, and batches of images can be received by calling the `flow()` function:

```
imagedatagen.flow(x_train, y_train, batch_size=32)
```

Finally, call the `fit_generator()` function instead of calling the `fit()` function on the model:

```
fit_generator(imagedatagen, samples_per_epoch=len(X_train), epochs=200)
```

Let's look at some examples to understand how the image augmentation API in Keras works. We will use the MNIST handwritten digit recognition task in these examples.

Let's begin by taking a look at the first nine images in the training dataset:

```
#Plot images
from keras.datasets import mnist
from matplotlib import pyplot
#loading data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
#creating a grid of 3x3 images
for i in range(0, 9):
  pyplot.subplot(330 + 1 + i)
  pyplot.imshow(X_train[i], cmap=pyplot.get_cmap('gray'))
#Displaying the plot
pyplot.show()
```

The following code snippet creates augmented images from the CIFAR-10 dataset. We will add these images to the training set of the last example and see how the classification accuracy increases:

```
from keras.preprocessing.image import ImageDataGenerator
# creating and configuring augmented image generator
datagen_train = ImageDataGenerator(
 width_shift_range=0.1, # shifting randomly images horizontally (10% of
total width)
 height_shift_range=0.1, # shifting randomly images vertically (10% of
total height)
 horizontal_flip=True) # flipping randomly images horizontally
# creating and configuring augmented image generator
```

```
datagen_valid = ImageDataGenerator(
 width_shift_range=0.1, # shifting randomly images horizontally (10% of
total width)
 height_shift_range=0.1, # shifting randomly images vertically (10% of
total height)
 horizontal_flip=True) # flipping randomly images horizontally
# fitting augmented image generator on data
datagen_train.fit(x_train)
datagen_valid.fit(x_valid)
```

Summary

We began this chapter by briefly looking into the history of CNNs. We introduced you to the implementation of visualizing images.

We studied image classification with the help of a practical example, using all the principles we learned about in the chapter. Finally, we learned how image augmentation helps us avoid overfitting and studied the various other features provided by image augmentation.

In the next chapter, we will learn how to build a simple image classifier CNN model from scratch.

3

Build Your First CNN and Performance Optimization

A **convolutional neural network** (**CNN**) is a type of **feed-forward neural network** (**FNN**) in which the connectivity pattern between its neurons is inspired by an animal's visual cortex. In the last few years, CNNs have demonstrated superhuman performance in image search services, self-driving cars, automatic video classification, voice recognition, and **natural language processing** (**NLP**).

Considering these motivations, in this chapter, we will construct a simple CNN model for image classification from scratch, followed by some theoretical aspects, such as convolutional and pooling operations. Then we will discuss how to tune hyperparameters and optimize the training time of CNNs for improved classification accuracy. Finally, we will build the second CNN model by considering some best practices. In a nutshell, the following topics will be covered in this chapter:

- CNN architectures and drawbacks of DNNs
- The convolution operations and pooling layers
- Creating and training a CNN for image classification
- Model performance optimization
- Creating an improved CNN for optimized performance

CNN architectures and drawbacks of DNNs

In `Chapter 2`, *Introduction to Convolutional Neural Networks*, we discussed that a regular multilayer perceptron works fine for small images (for example, MNIST or CIFAR-10). However, it breaks down for larger images because of the huge number of parameters it requires. For example, a 100 × 100 image has 10,000 pixels, and if the first layer has just 1,000 neurons (which already severely restricts the amount of information transmitted to the next layer), this means 10 million connections; and that is just for the first layer.

CNNs solve this problem using partially connected layers. Because consecutive layers are only partially connected and because it heavily reuses its weights, a CNN has far fewer parameters than a fully connected DNN, which makes it much faster to train, reduces the risk of overfitting, and requires much less training data. Moreover, when a CNN has learned a kernel that can detect a particular feature, it can detect that feature anywhere on the image. In contrast, when a DNN learns a feature in one location, it can detect it only in that particular location.

Since images typically have very repetitive features, CNNs are able to generalize much better than DNNs for image processing tasks such as classification, using fewer training examples. Importantly, a DNN has no prior knowledge of how pixels are organized; it does not know that nearby pixels are close. A CNN's architecture embeds this prior knowledge. Lower layers typically identify features in small areas of the images, while higher layers combine the lower-level features into larger features. This works well with most natural images, giving CNNs a decisive head start compared to DNNs:

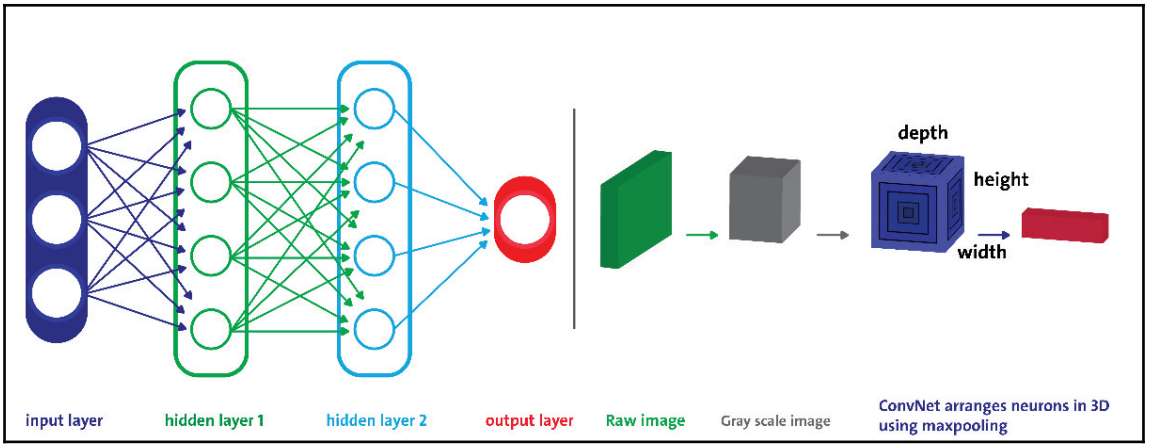


Figure 1: Regular DNN versus CNN, where each layer has neurons arranged in 3D

For example, in *Figure 1*, on the left, you can see a regular three-layer neural network. On the right, a ConvNet arranges its neurons in three dimensions (width, height, and depth) as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. The red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be three (red, green, and blue channels). Therefore, all the multilayer neural networks we looked at had layers composed of a long line of neurons, and we had to flatten input images or data to 1D before feeding them to the neural network.

However, what happens once you try to feed them a 2D image directly? The answer is that in CNNs, each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs. We will see examples of this in upcoming sections. Another important fact is that all the neurons in a feature map share the same parameters, so it dramatically reduces the number of parameters in the model; but more importantly, it means that once the CNN has learned to recognize a pattern in one location, it can recognize it in any other location.

In contrast, once a regular DNN has learned to recognize a pattern in one location, it can recognize it only in that particular location. In multilayer networks such as MLP or DBN, the outputs of all neurons of the input layer are connected to each neuron in the hidden layer, and then the output will again act as the input to the fully connected layer. In CNN networks, the connection scheme that defines the convolutional layer is significantly different. The convolutional layer is the main type of layer in a CNN, where each neuron is connected to a certain region of the input area called the **receptive field**.

In a typical CNN architecture, a few convolutional layers are connected in a cascade style. Each layer is followed by a **Rectified Linear Unit** (**ReLU**) layer, then a pooling layer, then one or more convolutional layers (+ReLU), then another pooling layer, and finally one or more fully connected layers. Pretty much depending on problem type, the network might be deep though. The output from each convolution layer is a set of objects called **feature maps**, generated by a single kernel filter. Then the feature maps can be used to define a new input to the next layer.

Each neuron in a CNN network produces an output, followed by an activation threshold, which is proportional to the input and not bound:

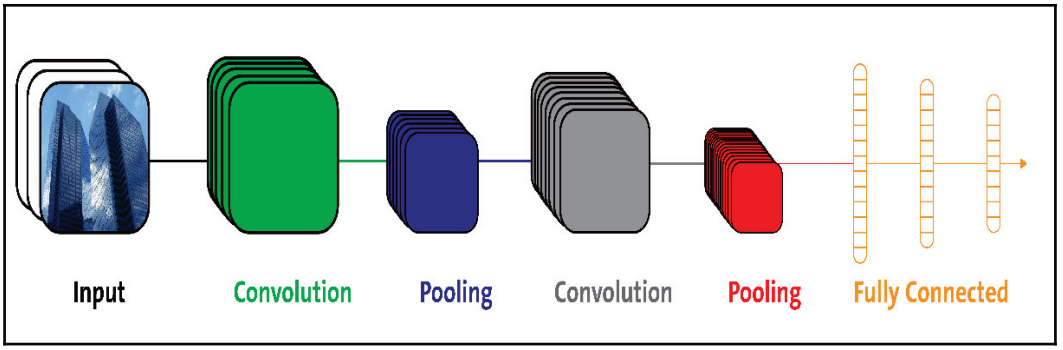
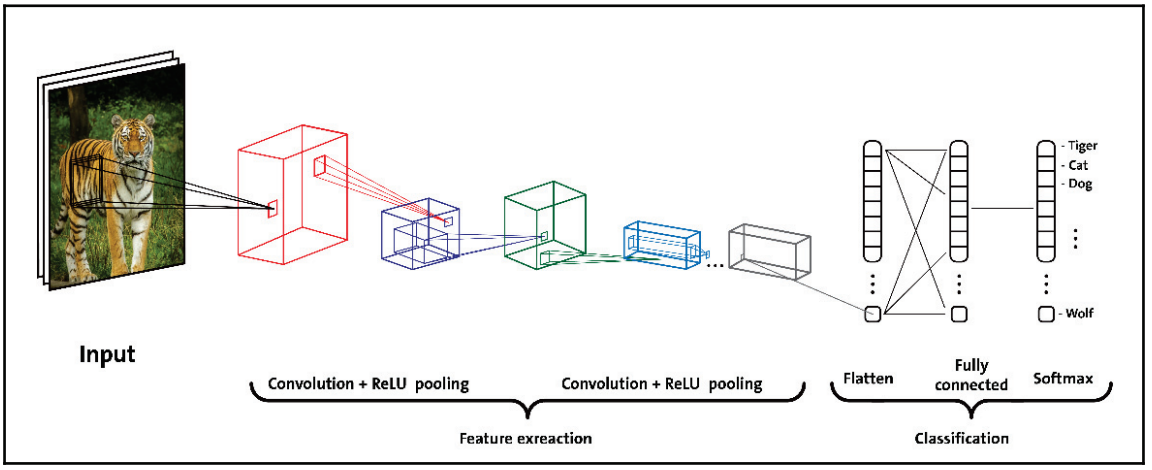


Figure 2: A conceptual architecture of a CNN

As you can see in *Figure 2*, the pooling layers are usually placed after the convolutional layers (for example, between two convolutional layers). A pooling layer into subregions then divides the convolutional region. Then, a single representative value is selected, using either a max-pooling or an average pooling technique, to reduce the computational time of subsequent layers. This way, a CNN can be thought of as a feature extractor. To understand this more clearly, refer to the following figure:



In this way, the robustness of the feature with respect to its spatial position is increased too. To be more specific, when feature maps are used as image properties and pass through the grayscale image, it gets smaller and smaller as it progresses through the network; but it also typically gets deeper and deeper, as more feature maps will be added.

We've already discussed the limitations of such FFNN - that is, a very high number of neurons would be necessary, even in a shallow architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters.

Convolutional operations

A convolution is a mathematical operation that slides one function over another and measures the integral of their pointwise multiplication. It has deep connections with the Fourier transformation and the Laplace transformation and is heavily used in signal processing. Convolutional layers actually use cross-correlations, which are very similar to convolutions.

In mathematics, convolution is a mathematical operation on two functions that produces a third function—that is, the modified (convoluted) version of one of the original functions. The resulting function gives in integral of the pointwise multiplication of the two functions as a function of the amount that one of the original functions is translated. Interested readers can refer to this URL for more information: `https://en.wikipedia.org/wiki/Convolution`.

Thus, the most important building block of a CNN is the convolutional layer. Neurons in the first convolutional layer are not connected to every single pixel in the input image (that is, like FNNs—for example, MLP and DBN) but only to pixels in their receptive fields. See *Figure 3*. In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer:

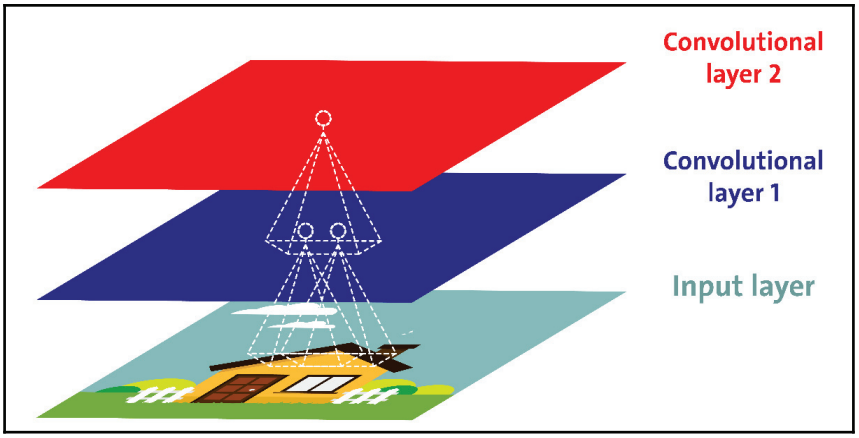


Figure 3: Each convolutional neuron processes data only for its receptive field

In `Chapter 2`, *Introduction to Convolutional Neural Networks*, we have seen that all multilayer neural networks (for example, MLP) have layers composed of so many neurons, and we have to flatten input images to 1D before feeding them to the neural network. Instead, in a CNN, each layer is represented in 2D, which makes it easier to match neurons with their corresponding inputs.

The receptive fields concept is used by CNNs to exploit spatial locality by enforcing a local connectivity pattern between neurons of adjacent layers.

This architecture allows the network to concentrate on low-level features in the first hidden layer, and then assemble them into higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

Finally, it not only requires a low number of neurons but also reduces the number of trainable parameters significantly. For example, regardless of image size, building regions of size 5 x 5, each with the same-shared weights, requires only 25 learnable parameters. In this way, it resolves the vanishing or exploding gradients problem in training traditional multilayer neural networks with many layers by using backpropagation.

Pooling, stride, and padding operations

Once you've understood how convolutional layers work, the pooling layers are quite easy to grasp. A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth. You may alternatively pool over the depth dimension, as we will see next, in which case the image's spatial dimensions (for example, height and width) remain unchanged but the number of channels is reduced. Let's see a formal definition of pooling layers from the well-known TensorFlow website:

> *"The pooling ops sweep a rectangular window over the input tensor, computing a reduction operation for each window (average, max, or max with argmax). Each pooling op uses rectangular windows of size called ksize separated by offset strides. For example, if strides are all ones, every window is used, if strides are all twos, every other window is used in each dimension, and so on."*

Therefore, in summary, just like convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field. However, we must define its size, the stride, and the padding type. So in summary, the output can be computed as follows:

```
output[i] = reduce(value[strides * i:strides * i + ksize]),
```

Here, the indices also take the padding values into consideration.

A pooling neuron has no weights. Therefore, all it does is aggregate the inputs using an aggregation function such as max or mean.

In other words, the goal of using pooling is to subsample the input image in order to reduce the computational load, memory usage, and number of parameters. This helps to avoid overfitting in the training stage. Reducing the input image size also makes the neural network tolerate a little bit of image shift. The spatial semantics of the convolution ops depend on the padding scheme chosen.

Padding is an operation to increase the size of the input data. In the case of one-dimensional data, you just append/prepend the array with a constant; in two-dimensional data, you surround the matrix with these constants. In n-dimensional, you surround your n-dimensional hypercube with the constant. In most of the cases, this constant is zero and it is called **zero padding**:

- **VALID padding**: Only drops the rightmost columns (or bottommost rows)
- **SAME padding**: Tries to pad evenly left and right, but if the number of columns to be added is odd, it will add the extra column to the right, as is the case in this example

Let's explain the preceding definition graphically, in the following figure. If we want a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs, as shown in the diagram. This is called **SAME** or **zero padding**.

The term **SAME** means that the output feature map has the same spatial dimensions as the input feature map.

On the other hand, zero padding is introduced to make the shapes match as needed, equally on every side of the input map. **VALID** means no padding and only drops the rightmost columns (or bottommost rows):

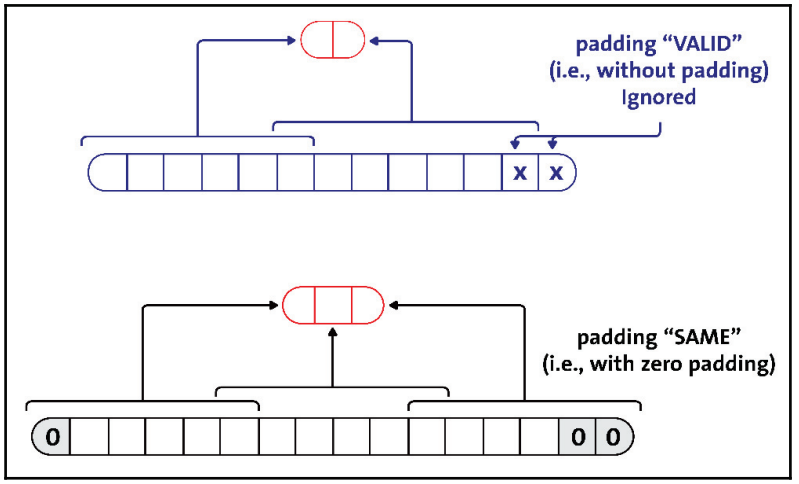


Figure 4: SAME versus VALID padding with CNN

In the following example (*Figure 5*), we use a 2 × 2 pooling kernel and a stride of 2 with no padding. Only the **max** input value in each kernel makes it to the next layer since the other inputs are dropped (we will see this later on):

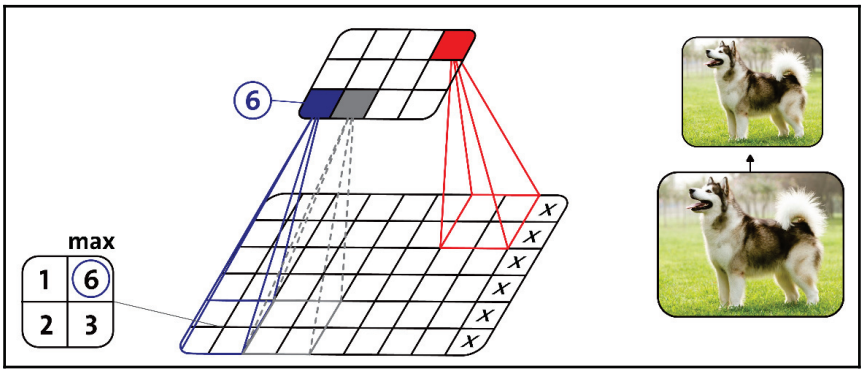


Figure 5: An example using max pooling, that is, subsampling

Fully connected layer

At the top of the stack, a regular fully connected layer (also known as **FNN** or **dense layer**) is added; it acts similar to an MLP, which might be composed of a few fully connected layers (+ReLUs). The final layer outputs (for example, softmax) the prediction. An example is a softmax layer that outputs estimated class probabilities for a multiclass classification.

Fully connected layers connect every neuron in one layer to every neuron in another layer. Although fully connected FNNs can be used to learn features as well as classify data, it is not practical to apply this architecture to images.

Convolution and pooling operations in TensorFlow

Now that we have seen how convolutional and pooling operations are performed theoretically, let's see how we can perform these operation hands-on using TensorFlow. So let's get started.

Applying pooling operations in TensorFlow

Using TensorFlow, a subsampling layer can normally be represented by a `max_pool` operation by maintaining the initial parameters of the layer. For `max_pool`, it has the following signature in TensorFlow:

```
tf.nn.max_pool(value, ksize, strides, padding, data_format, name)
```

Now let's learn how to create a function that utilizes the preceding signature and returns a tensor with type `tf.float32`, that is, the max pooled output tensor:

```
import tensorflow as tf

def maxpool2d(x, k=2):
   return tf.nn.max_pool(x,
               ksize=[1, k, k, 1],
               strides=[1, k, k, 1],
               padding='SAME')
```

In the preceding code segment, the parameters can be described as follows:

- `value`: This is a 4D tensor of `float32` elements and shape (batch length, height, width, and channels)
- `ksize`: A list of integers representing the window size on each dimension
- `strides`: The step of the moving windows on each dimension
- `data_format`: `NHWC`, `NCHW`, and `NCHW_VECT_C` are supported
- `ordering`: `NHWC` or `NCHW`
- `padding`: `VALID` or `SAME`

However, depending upon the layering structures in a CNN, there are other pooling operations supported by TensorFlow, as follows:

- `tf.nn.avg_pool`: This returns a reduced tensor with the average of each window
- `tf.nn.max_pool_with_argmax`: This returns the `max_pool` tensor and a tensor with the flattened index of `max_value`
- `tf.nn.avg_pool3d`: This performs an `avg_pool` operation with a cubic-like
- window; the input has an added depth
- `tf.nn.max_pool3d`: This performs the same function as (...) but applies the max operation

Now let's see a concrete example of how the padding thing works in TensorFlow. Suppose we have an input image `x` with shape `[2, 3]` and one channel. Now we want to see the effect of both `VALID` and `SAME` paddings:

- `valid_pad`: Max pool with 2 x 2 kernel, stride 2, and `VALID` padding
- `same_pad`: Max pool with 2 x 2 kernel, stride 2, and `SAME` padding

Let's see how we can attain this in Python and TensorFlow. Suppose we have an input image of shape `[2, 4]`, which is one channel:

```
import tensorflow as tf
x = tf.constant([[2., 4., 6., 8.,],
                 [10., 12., 14., 16.]])
```

Now let's give it a shape accepted by `tf.nn.max_pool`:

```
x = tf.reshape(x, [1, 2, 4, 1])
```

If we want to apply the `VALID` padding with the max pool with a 2 x 2 kernel, stride 2:

```
VALID = tf.nn.max_pool(x, [1, 2, 2, 1], [1, 2, 2, 1], padding='VALID')
```

On the other hand, using the max pool with a 2 x 2 kernel, stride 2 and `SAME` padding:

```
SAME = tf.nn.max_pool(x, [1, 2, 2, 1], [1, 2, 2, 1], padding='SAME')
```

For `VALID` padding, since there is no padding, the output shape is `[1, 1]`. However, for the `SAME` padding, since we pad the image to the shape `[2, 4]` (with `-inf`) and then apply the max pool, the output shape is `[1, 2]`. Let's validate them:

```
print(VALID.get_shape())
print(SAME.get_shape())
```

```
>>>
(1, 1, 2, 1)
(1, 1, 2, 1)
```

Convolution operations in TensorFlow

TensorFlow provides a variety of methods for convolution. The canonical form is applied by the `conv2d` operation. Let's have a look at the usage of this operation:

```
conv2d(
     input,
     filter,
     strides,
     padding,
     use_cudnn_on_gpu=True,
     data_format='NHWC',
     dilations=[1, 1, 1, 1],
     name=None
 )
```

The parameters we use are as follows:

- `input`: The operation will be applied to this original tensor. It has a definite format of four dimensions, and the default dimension order is shown next.
- `filter`: This is a tensor representing a kernel or filter. It has a very generic method: (`filter_height`, `filter_width`, `in_channels`, and `out_channels`).
- `strides`: This is a list of four `int` tensor datatypes, which indicate the sliding windows for each dimension.
- `padding`: This can be `SAME` or `VALID`. `SAME` will try to conserve the initial tensor dimension, but `VALID` will allow it to grow if the output size and padding are computed. We will see later how to perform padding along with the pooling layers.
- `use_cudnn_on_gpu`: This indicates whether to use the `CUDA GPU CNN` library to accelerate calculations.
- `data_format`: This specifies the order in which data is organized (`NHWC` or `NCWH`).
- `dilations`: This signifies an optional list of `ints`. It defaults to (1, 1, 1, 1). 1D tensor of length 4. The dilation factor for each dimension of input. If it is set to k > 1, there will be k-1 skipped cells between each filter element on that dimension. The dimension order is determined by the value of `data_format`; see the preceding code example for details. Dilations in the batch and depth dimensions must be 1.
- `name`: A name for the operation (optional).

The following is an example of a convolutional layer. It concatenates a convolution, adds a bias parameter sum, and finally returns the activation function we have chosen for the whole layer (in this case, the ReLU operation, which is a frequently used one):

```
def conv_layer(data, weights, bias, strides=1):
   x = tf.nn.conv2d(x,
               weights,
               strides=[1, strides, strides, 1],
               padding='SAME')
   x = tf.nn.bias_add(x, bias)
   return tf.nn.relu(x)
```

Here, x is the 4D tensor input (batch size, height, width, and channel). TensorFlow also offers a few other kinds of convolutional layers. For example:

- `tf.layers.conv1d()` creates a convolutional layer for 1D inputs. This is useful, for example, in NLP, where a sentence may be represented as a 1D array of words, and the receptive field covers a few neighboring words.
- `tf.layers.conv3d()` creates a convolutional layer for 3D inputs.
- `tf.nn.atrous_conv2d()` creates an a trous convolutional layer (*a* tro*us* is French for with holes). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros. For example, a 1 × 3 filter equal to (1, 2, 3) may be dilated with a dilation rate of 4, resulting in a dilated filter (1, 0, 0, 0, 2, 0, 0, 0, 3). This allows the convolutional layer to have a larger receptive field at no computational price and using no extra parameters.
- `tf.layers.conv2d_transpose ()` creates a transpose convolutional layer, sometimes called a **deconvolutional layer,** which up-samples an image. It does so by inserting zeros between the inputs, so you can think of this as a regular convolutional layer using a fractional stride.
- `tf.nn.depthwise_conv2d()` creates a depth-wise convolutional layer that applies every filter to every individual input channel independently. Thus, if there are f_n filters and $f_{n'}$ input channels, then this will output $f_n \times f_{n'}$ feature maps.
- `tf.layers.separable_conv2d()` creates a separable convolutional layer that first acts like a depth-wise convolutional layer and then applies a 1 × 1 convolutional layer to the resulting feature maps. This makes it possible to apply filters to arbitrary sets of inputs channels.

Training a CNN

In the previous section, we have seen how to construct a CNN and apply different operations on its different layers. Now when it comes to training a CNN, it is much trickier as it needs a lot of considerations to control those operations such as applying appropriate activation function, weight and bias initialization, and of course, using optimizers intelligently.

There are also some advanced considerations such as hyperparameter tuning for optimized too. However, that will be discussed in the next section. We first start our discussion with weight and bias initialization.

Weight and bias initialization

One of the most common initialization techniques in training a DNN is random initialization. The idea of using random initialization is just sampling each weight from a normal distribution of the input dataset with low deviation. Well, a low deviation allows you to bias the network towards the simple 0 solutions.

But what does it mean? The thing is that, the initialization can be completed without the bad repercussions of actually initializing the weights to 0. Secondly, Xavier initialization is often used to train CNNs. It is similar to random initialization but often turns out to work much better. Now let me explain the reason for this:

- Imagine that you initialize the network weights randomly but they turn out to start too small. Then the signal shrinks as it passes through each layer until it is too tiny to be useful.
- On the other hand, if the weights in a network start too large, then the signal grows as it passes through each layer until it is too massive to be useful.

The good thing is that using Xavier initialization makes sure the weights are just right, keeping the signal in a reasonable range of values through many layers. In summary, it can automatically determine the scale of initialization based on the number of input and output neurons.

Interested readers should refer to this publication for detailed information: Xavier Glorot and Yoshua Bengio, *Understanding the difficulty of training deep FNNs*, Proceedings of the 13th International Conference on **Artificial Intelligence and Statistics** (**AISTATS**) 2010, Chia Laguna Resort, Sardinia, Italy. Volume 9 of JMLR: W&CP.

Finally, you may ask an intelligent question, *Can't I get rid of the random initialization while training a regular DNN (for example, MLP or DBN)*? Well, recently, some researchers have been talking about random orthogonal matrix initializations that perform better than just any random initialization for training DNNs:

- **When it comes to initializing the biases**, it is possible and common to initialize the biases to be zero since the asymmetry breaking is provided by the small random numbers in the weights. Setting the biases to a small constant value such as 0.01 for all biases ensures that all ReLU units can propagate some gradient. However, it neither performs well nor does consistent improvement. Therefore, sticking with zero is recommended.

Regularization

There are several ways of controlling training of CNNs to prevent overfitting in the training phase. For example, L2/L1 regularization, max norm constraints, and drop out:

- **L2 regularization**: This is perhaps the most common form of regularization. It can be implemented by penalizing the squared magnitude of all parameters directly in the objective. For example, using the gradient descent parameter update, L2 regularization ultimately means that every weight is decayed linearly: *W* += -lambda * *W* towards zero.
- **L1 regularization**: This is another relatively common form of regularization, where for each weight w we add the term $\lambda|w|$ to the objective. However, it is also possible to possible to combine the L1 regularization with the L2 regularization: $\lambda 1|w|+\lambda 2w2$, which is commonly known as **Elastic-net regularization**.
- **Max-norm constraints**: Another form of regularization is to enforce an absolute upper bound on the magnitude of the weight vector for every neuron and use projected gradient descent to enforce the constraint.

Finally, dropout is an advanced variant of regularization, which will be discussed later in this chapter.

Activation functions

The activation ops provide different types of nonlinearities for use in neural networks. These include smooth nonlinearities, such as `sigmoid`, `tanh`, `elu`, `softplus`, and `softsign`. On the other hand, some continuous but not-everywhere-differentiable functions that can be used are `relu`, `relu6`, `crelu`, and `relu_x`. All activation ops apply component-wise and produce a tensor of the same shape as the input tensor. Now let us see how to use a few commonly used activation functions in TensorFlow syntax.

Using sigmoid

In TensorFlow, the signature `tf.sigmoid(x, name=None)` computes sigmoid of `x` element-wise using *y = 1 / (1 + exp(-x))* and returns a tensor with the same type `x`. Here is the parameter description:

- `x`: A tensor. This must be one of the following types: `float32`, `float64`, `int32`, `complex64`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Using tanh

In TensorFlow, the signature `tf.tanh(x, name=None)` computes a hyperbolic tangent of `x` element-wise and returns a tensor with the same type `x`. Here is the parameter description:

- `x`: A tensor or sparse. This is a tensor with type `float`, `double`, `int32`, `complex64`, `int64`, or `qint32`.
- `name`: A name for the operation (optional).

Using ReLU

In TensorFlow, the signature `tf.nn.relu(features, name=None)` computes a rectified linear using `max(features, 0)` and returns a tensor having the same type as features. Here is the parameter description:

- `features`: A tensor. This must be one of the following types: `float32`, `float64`, `int32`, `int64`, `uint8`, `int16`, `int8`, `uint16`, and `half`.
- `name`: A name for the operation (optional).

For more on how to use other activation functions, please refer to the TensorFlow website. Up to this point, we have the minimal theoretical knowledge to build our first CNN network for making a prediction.

Building, training, and evaluating our first CNN

In the next section, we will look at how to classify and distinguish between dogs from cats based on their raw images. We will also look at how to implement our first CNN model to deal with the raw and color image having three channels. This network design and implementation are not straightforward; TensorFlow low-level APIs will be used for this. However, do not worry; later in this chapter, we will see another example of implementing a CNN using TensorFlow's high-level contrib API. Before we formally start, a short description of the dataset is a mandate.

Dataset description

For this example, we will use the dog versus cat dataset from Kaggle that was provided for the infamous Dogs versus Cats classification problem as a playground competition with kernels enabled. The dataset can be downloaded from `https://www.kaggle.com/c/dogs-vs-cats-redux-kernels-edition/data`.

The train folder contains 25,000 images of dogs and cats. Each image in this folder has the label as part of the filename. The test folder contains 12,500 images, named according to a numeric ID. For each image in the test set, you should predict a probability that the image is a dog (1 = dog, 0 = cat); that is, a binary classification problem. For this example, there are three Python scripts.

Step 1 – Loading the required packages

Here we import the required packages and libraries. Note that depending upon the platform, your imports might be different:

```
import time
import math
import random
import os
import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import Preprocessor
import cv2
import LayersConstructor
from sklearn.metrics import confusion_matrix
from datetime import timedelta
from sklearn.metrics.classification import accuracy_score
from sklearn.metrics import precision_recall_fscore_support
```

Step 2 – Loading the training/test images to generate train/test set

We set the number of color channels as 3 for the images. In the previous section, we have seen that it should be 1 for grayscale images:

```
num_channels = 3
```

For the simplicity, we assume the image dimensions should be squares only. Let's set the size to be `128`:

```
img_size = 128
```

Now that we have the image size (that is, `128`) and the number of the channel (that is, 3), the size of the image when flattened to a single dimension would be the multiplication of the image dimension and the number of channels, as follows:

```
img_size_flat = img_size * img_size * num_channels
```

Note that, at a later stage, we might need to reshape the image for the max pooling and convolutional layers, so we need to reshape the image. For our case, it would be the tuple with height and width of images used to reshape arrays:

```
img_shape = (img_size, img_size)
```

We should have explicitly defined the labels (that is, classes) since we only have the raw color image, and so the images do not have the labels like other numeric machine learning dataset, have. Let's explicitly define the class info as follows:

```
classes = ['dogs', 'cats']
num_classes = len(classes)
```

We need to define the batch size that needs to be trained on our CNN model later on:

```
batch_size = 14
```

Note that we also can define what portion of the training set will be used as the validation split. Let's assume that 16% will be used, for simplicity:

```
validation_size = 0.16
```

One important thing to set is how long to wait after the validation loss stops improving before terminating the training. We should use none if we do not want to implement early stopping:

```
early_stopping = None
```

Now, download the dataset and you have to do one thing manually: separate the images of dogs and cats and place them in two separate folders. To be more specific, suppose you put your training set under the path `/home/DoG_CaT/data/train/`. In the train folder, create two separate folders `dogs` and `cats` but only show the path to `DoG_CaT/data/train/`. We also assume that our test set is in the `/home/DoG_CaT/data/test/` directory. In addition, you can define the checkpoint directory where the logs and model checkpoint files will be written:

```
train_path = '/home/DoG_CaT/data/train/'
test_path = '/home/DoG_CaT/data/test/'
checkpoint_dir = "models/"
```

Then we start reading the training set and prepare it for the CNN model. For processing the test and train set, we have another script `Preprocessor.py`. Nonetheless, it would be better to prepare the test set as well**:**

```
data = Preprocessor.read_train_sets(train_path, img_size, classes,
validation_size=validation_size)
```

The preceding line of code reads the raw images of cats and dogs and creates the training set. The `read_train_sets()` function goes as follows:

```
def read_train_sets(train_path, image_size, classes, validation_size=0):
  class DataSets(object):
      pass
      data_sets = DataSets()
      images, labels, ids, cls = load_train(train_path, image_size,
classes)
      images, labels, ids, cls = shuffle(images, labels, ids, cls)
      if isinstance(validation_size, float):
          validation_size = int(validation_size * images.shape[0])
```

```
            validation_images = images[:validation_size]
            validation_labels = labels[:validation_size]
            validation_ids = ids[:validation_size]
            validation_cls = cls[:validation_size]
            train_images = images[validation_size:]
            train_labels = labels[validation_size:]
            train_ids = ids[validation_size:]
            train_cls = cls[validation_size:]
            data_sets.train = DataSet(train_images, train_labels, train_ids,
train_cls)
            data_sets.valid = DataSet(validation_images, validation_labels,
validation_ids, validation_cls)
  return data_sets
```

In the previous code segment, we have used the method `load_train()` to load the images which is an instance of a class called `DataSet`:

```
def load_train(train_path, image_size, classes):
    images = []
    labels = []
    ids = []
    cls = []

    print('Reading training images')
    for fld in classes:
        index = classes.index(fld)
        print('Loading {} files (Index: {})'.format(fld, index))
        path = os.path.join(train_path, fld, '*g')
        files = glob.glob(path)
        for fl in files:
            image = cv2.imread(fl)
            image = cv2.resize(image, (image_size, image_size),
cv2.INTER_LINEAR)
            images.append(image)
            label = np.zeros(len(classes))
            label[index] = 1.0
            labels.append(label)
            flbase = os.path.basename(fl)
            ids.append(flbase)
            cls.append(fld)
    images = np.array(images)
    labels = np.array(labels)
    ids = np.array(ids)
    cls = np.array(cls)
    return images, labels, ids, cls
```

The `DataSet` class, which is used to generate the batches of the training set, is as follows:

```
class DataSet(object):
  def next_batch(self, batch_size):
    """Return the next `batch_size` examples from this data set."""
    start = self._index_in_epoch
    self._index_in_epoch += batch_size
    if self._index_in_epoch > self._num_examples:
      # Finished epoch
      self._epochs_completed += 1
      start = 0
      self._index_in_epoch = batch_size
      assert batch_size <= self._num_examples
    end = self._index_in_epoch
    return self._images[start:end], self._labels[start:end],
self._ids[start:end], self._cls[start:end]
```

Then, similarly, we prepare the test set from the test images that are mixed (dogs and cats):

```
test_images, test_ids = Preprocessor.read_test_set(test_path, img_size)
```

We have the `read_test_set()` function for ease, as follows:

```
def read_test_set(test_path, image_size):
  images, ids  = load_test(test_path, image_size)
  return images, ids
```

Now, similar to the training set, we have a dedicated function called `load_test ()` for loading the test set, which goes as follows:

```
def load_test(test_path, image_size):
  path = os.path.join(test_path, '*g')
  files = sorted(glob.glob(path))

  X_test = []
  X_test_id = []
  print("Reading test images")
  for fl in files:
      flbase = os.path.basename(fl)
      img = cv2.imread(fl)
      img = cv2.resize(img, (image_size, image_size), cv2.INTER_LINEAR)
      X_test.append(img)
      X_test_id.append(flbase)
  X_test = np.array(X_test, dtype=np.uint8)
  X_test = X_test.astype('float32')
  X_test = X_test / 255
  return X_test, X_test_id
```

Well done! We can now see some randomly selected images. For this, we have the helper function called `plot_images()`; it creates a figure with 3 x 3 sub-plots. So, all together, nine images will be plotted, along with their true label. It goes as follows:

```
def plot_images(images, cls_true, cls_pred=None):
    if len(images) == 0:
        print("no images to show")
        return
    else:
        random_indices = random.sample(range(len(images)), min(len(images),
9))
        images, cls_true  = zip(*[(images[i], cls_true[i]) for i in
random_indices])
    fig, axes = plt.subplots(3, 3)
    fig.subplots_adjust(hspace=0.3, wspace=0.3)
    for i, ax in enumerate(axes.flat):
        # Plot image.
        ax.imshow(images[i].reshape(img_size, img_size, num_channels))
        if cls_pred is None:
            xlabel = "True: {0}".format(cls_true[i])
        else:
            xlabel = "True: {0}, Pred: {1}".format(cls_true[i],
cls_pred[i])
        ax.set_xlabel(xlabel)
        ax.set_xticks([])
        ax.set_yticks([])
    plt.show()
```

Let's get some random images and their labels from the train set:

```
images, cls_true  = data.train.images, data.train.cls
```

Finally, we plot the images and labels using our helper-function in the preceding code:

```
plot_images(images=images, cls_true=cls_true)
```

The preceding line of code generates the true labels of the images that are randomly selected:

Figure 6: The true labels of the images that are randomly selected

Finally, we can print the dataset statistics:

```
print("Size of:")
print("  - Training-set:tt{}".format(len(data.train.labels)))
print("  - Test-set:tt{}".format(len(test_images)))
print("  - Validation-set:t{}".format(len(data.valid.labels)))

>>>
Reading training images
 Loading dogs files (Index: 0)
 Loading cats files (Index: 1)
 Reading test images
 Size of:
 - Training-set: 21000
 - Test-set: 12500
 - Validation-set: 4000
```

Step 3- Defining CNN hyperparameters

Now that we have the training and test set, it's time to define the hyperparameters for the CNN model before we start constructing. In the first and the second convolutional layers, we define the width and height of each filter, that is, `3`, where the number of filters is `32`:

```
filter_size1 = 3
num_filters1 = 32
filter_size2 = 3
num_filters2 = 32
```

The third convolutional layer has equal dimensions but twice the filters; that is, `64` filters:

```
filter_size3 = 3
num_filters3 = 64
```

The last two layers are fully connected layers, specifying the number of neurons:

```
fc_size = 128
```

Now let's make the training slower for more intensive training by setting a lower value of the learning rate, as follows:

```
learning_rate=1e-4
```

Step 4 – Constructing the CNN layers

Once we have defined the CNN hyperparameters, the next task is to implement the CNN network. As you can guess, for our task, we will construct a CNN network having three convolutional layers, a flattened layer and two fully connected layers (refer to `LayersConstructor.py`). Moreover, we need to define the weight and the bias as well. Furthermore, we will have implicit max-pooling layers too. At first, let's define the weight. In the following, we have the `new_weights()` method that asks for the image shape and returns the truncated normal shapes:

```
def new_weights(shape):
    return tf.Variable(tf.truncated_normal(shape, stddev=0.05))
```

Then we define the biases using the `new_biases()` method:

```
def new_biases(length):
    return tf.Variable(tf.constant(0.05, shape=[length]))
```

Now let's define a method, `new_conv_layer()`, for constructing a convolutional layer. The method takes the input batch, number of input channels, filter size, and number of filters and it also uses the max pooling (if true, we use a 2 x 2 max pooling) to construct a new convolutional layer. The workflow of the method is as follows:

1. Define the shape of the filter weights for the convolution, which is determined by the TensorFlow API.
2. Create the new weights (that is, filters) with the given shape and new biases, one for each filter.
3. Create the TensorFlow operation for the convolution where the strides are set to 1 in all dimensions. The first and last stride must always be 1, because the first is for the image-number and the last is for the input channel. For example, strides= (1, 2, 2, 1) would mean that the filter is moved two pixels across the *x* axis and *y* axis of the image.
4. Add the biases to the results of the convolution. Then a bias-value is added to each filter-channel.
5. It then uses the pooling to downsample the image resolution. This is 2 x 2 max pooling, which means that we consider 2 x 2 windows and select the largest value in each window. Then we move two pixels to the next window.
6. ReLU is then used to calculate the *max(x, 0)* for each input pixel *x*. As stated earlier, a ReLU is normally executed before the pooling, but since `relu(max_pool(x)) == max_pool(relu(x))` we can save 75% of the relu-operations by max-pooling first.
7. Finally, it returns both the resulting layer and the filter-weights because we will plot the weights later.

Now we define a function to construct the convolutional layer to be used:

```
def new_conv_layer(input,  num_input_channels, filter_size, num_filters,
use_pooling=True):
    shape = [filter_size, filter_size, num_input_channels, num_filters]
    weights = new_weights(shape=shape)
    biases = new_biases(length=num_filters)
    layer = tf.nn.conv2d(input=input,
                         filter=weights,
                         strides=[1, 1, 1, 1],
                         padding='SAME')
    layer += biases
    if use_pooling:
        layer = tf.nn.max_pool(value=layer,
                               ksize=[1, 2, 2, 1],
                               strides=[1, 2, 2, 1],
```

```
                                padding='SAME')
    layer = tf.nn.relu(layer)
    return layer, weights
```

The next task is to define the flattened layer:

1. Get the shape of the input layer.
2. The number of features is `img_height * img_width * num_channels`. The `get_shape()` function TensorFlow is used to calculate this.
3. It will then reshape the layer to (`num_images` and `num_features`). We just set the size of the second dimension to `num_features` and the size of the first dimension to -1, which means the size in that dimension is calculated so the total size of the tensor is unchanged from the reshaping.
4. Finally, it returns both the flattened layer and the number of features.

The following code does exactly the same as described before `defflatten_layer(layer):`

```
    layer_shape = layer.get_shape()
    num_features = layer_shape[1:4].num_elements()
    layer_flat = tf.reshape(layer, [-1, num_features])
    return layer_flat, num_features
```

Finally, we need to construct the fully connected layers. The following function, `new_fc_layer()`, takes the input batches, number of batches, and number of outputs (that is, predicted classes) and it uses the ReLU. It then creates the weights and biases based on the methods we define earlier in this step. Finally, it calculates the layer as the matrix multiplication of the input and weights, and then adds the bias values:

```
def new_fc_layer(input, num_inputs, num_outputs, use_relu=True):
    weights = new_weights(shape=[num_inputs, num_outputs])
    biases = new_biases(length=num_outputs)
    layer = tf.matmul(input, weights) + biases
    if use_relu:
        layer = tf.nn.relu(layer)
    return layer
```

Step 5 – Preparing the TensorFlow graph

We now create the placeholders for the TensorFlow graph:

```
x = tf.placeholder(tf.float32, shape=[None, img_size_flat], name='x')
x_image = tf.reshape(x, [-1, img_size, img_size, num_channels])
y_true = tf.placeholder(tf.float32, shape=[None, num_classes],
name='y_true')
y_true_cls = tf.argmax(y_true, axis=1)
```

Step 6 – Creating a CNN model

Now we have the input; that is, `x_image` is ready to feed to the convolutional layer. We formally create a convolutional layer, followed by the max pooling:

```
layer_conv1, weights_conv1 =
    LayersConstructor.new_conv_layer(input=x_image,
                    num_input_channels=num_channels,
                    filter_size=filter_size1,
                    num_filters=num_filters1,
                    use_pooling=True)
```

We must have the second convolutional layer, where the input is the first convolutional layer, `layer_conv1`, followed by the max pooling:

```
layer_conv2, weights_conv2 =
    LayersConstructor.new_conv_layer(input=layer_conv1,
                    num_input_channels=num_filters1,
                    filter_size=filter_size2,
                    num_filters=num_filters2,
                    use_pooling=True)
```

We now have the third convolutional layer where the input is the output of the second convolutional layer, that is, `layer_conv2` followed by the max pooling:

```
layer_conv3, weights_conv3 =
    LayersConstructor.new_conv_layer(input=layer_conv2,
                    num_input_channels=num_filters2,
                    filter_size=filter_size3,
                    num_filters=num_filters3,
                    use_pooling=True)
```

Once the third convolutional layer is instantiated, we then instantiate the flattened layer as follows:

```
layer_flat, num_features = LayersConstructor.flatten_layer(layer_conv3)
```

Once we have flattened the images, they are ready to be fed to the first fully connected layer. We use the ReLU:

```
layer_fc1 = LayersConstructor.new_fc_layer(input=layer_flat,
                          num_inputs=num_features,
                          num_outputs=fc_size,
                          use_relu=True)
```

Finally, we have to have the second and the final fully connected layer where the input is the output of the first fully connected layer:

```
layer_fc2 = LayersConstructor.new_fc_layer(input=layer_fc1,
                          num_inputs=fc_size,
                          num_outputs=num_classes,
                          use_relu=False)
```

Step 7 – Running the TensorFlow graph to train the CNN model

The following steps are used to perform the training. The codes are self-explanatory, like the ones that we have already used in our previous examples. We use softmax to predict the classes by comparing them with true classes:

```
y_pred = tf.nn.softmax(layer_fc2)
y_pred_cls = tf.argmax(y_pred, axis=1)
cross_entropy =
tf.nn.softmax_cross_entropy_with_logits_v2(logits=layer_fc2,
labels=y_true)
```

We define the `cost` function and then the optimizer (Adam optimizer in this case). Then we compute the accuracy:

```
cost_op= tf.reduce_mean(cross_entropy)
optimizer =
tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost_op)
correct_prediction = tf.equal(y_pred_cls, y_true_cls)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Then we initialize all the ops using the `global_variables_initializer()` function from TensorFlow:

```
init_op = tf.global_variables_initializer()
```

Then we create and run the TensorFlow session to carry the training across the tensors:

```
session = tf.Session()
session.run(init_op)
```

We then feed out training data so that the batch size to 32 (see *Step 2*):

```
train_batch_size = batch_size
```

We maintain two lists to track the training and validation accuracy:

```
acc_list = []
val_acc_list = []
```

We then count the total number of iterations performed so far and create an empty list to keep track of all the iterations:

```
total_iterations = 0
iter_list = []
```

We formally start the training by invoking the `optimize()` function, which takes a number of iterations. It needs two:

- The `x_batch` of training examples that holds a batch of images and
- `y_true_batch`, the true labels for those images

It then converts the shape of each image from (`num` examples, rows, columns, depth) to (`num` examples, flattened image shape). After that, we put the batch into a `dict` for placeholder variables in the TensorFlow graph. Later on, we run the optimizer on the batch of training data.

Then, TensorFlow assigns the variables in `feed_dict_train` to the placeholder variables. Optimizer is then executed to print the status at end of each epoch. Finally, it updates the total number of iterations that we performed:

```
def optimize(num_iterations):
    global total_iterations
    best_val_loss = float("inf")
    patience = 0
    for i in range(total_iterations, total_iterations + num_iterations):
        x_batch, y_true_batch, _, cls_batch =
```

```
data.train.next_batch(train_batch_size)
        x_valid_batch, y_valid_batch, _, valid_cls_batch =
data.valid.next_batch(train_batch_size)
        x_batch = x_batch.reshape(train_batch_size, img_size_flat)
        x_valid_batch = x_valid_batch.reshape(train_batch_size,
img_size_flat)
        feed_dict_train = {x: x_batch, y_true: y_true_batch}
        feed_dict_validate = {x: x_valid_batch, y_true: y_valid_batch}
        session.run(optimizer, feed_dict=feed_dict_train)

        if i % int(data.train.num_examples/batch_size) == 0:
            val_loss = session.run(cost, feed_dict=feed_dict_validate)
            epoch = int(i / int(data.train.num_examples/batch_size))
            acc, val_acc = print_progress(epoch, feed_dict_train,
feed_dict_validate, val_loss)
            acc_list.append(acc)
            val_acc_list.append(val_acc)
            iter_list.append(epoch+1)
            if early_stopping:
                if val_loss < best_val_loss:
                    best_val_loss = val_loss
                    patience = 0
                else:
                    patience += 1
                if patience == early_stopping:
                    break
    total_iterations += num_iterations
```

We will show how our training went along in the next section.

Step 8 – Model evaluation

We have managed to finish the training. It is time to evaluate the model. Before, we start evaluating the model, let's implement some auxiliary functions for plotting the example errors and printing the validation accuracy. The `plot_example_errors()` takes two parameters. The first is `cls_pred`, which is an array of the predicted class-number for all images in the test set.

The second parameter, `correct`, is a `boolean` array to predict whether the predicted class is equal to `true` class for each image in the test set. At first, it gets the images from the test set that have been incorrectly classified. Then it gets the predicted and the true classes for those images, and finally it plots the first nine images with their classes (that is, predicted versus true labels):

```
def plot_example_errors(cls_pred, correct):
    incorrect = (correct == False)
    images = data.valid.images[incorrect]
    cls_pred = cls_pred[incorrect]
    cls_true = data.valid.cls[incorrect]
    plot_images(images=images[0:9], cls_true=cls_true[0:9],
cls_pred=cls_pred[0:9])
```

The second auxiliary function is called `print_validation_accuracy()`; it prints the validation accuracy. It allocates an array for the predicted classes, which will be calculated in batches and filled into this array, and then it calculates the predicted classes for the batches:

```
def print_validation_accuracy(show_example_errors=False,
show_confusion_matrix=False):
    num_test = len(data.valid.images)
    cls_pred = np.zeros(shape=num_test, dtype=np.int)
    i = 0
    while i < num_test:
        # The ending index for the next batch is denoted j.
        j = min(i + batch_size, num_test)
        images = data.valid.images[i:j, :].reshape(batch_size,
img_size_flat)
        labels = data.valid.labels[i:j, :]
        feed_dict = {x: images, y_true: labels}
        cls_pred[i:j] = session.run(y_pred_cls, feed_dict=feed_dict)
        i = j

    cls_true = np.array(data.valid.cls)
    cls_pred = np.array([classes[x] for x in cls_pred])
    correct = (cls_true == cls_pred)
    correct_sum = correct.sum()
    acc = float(correct_sum) / num_test

    msg = "Accuracy on Test-Set: {0:.1%} ({1} / {2})"
    print(msg.format(acc, correct_sum, num_test))

    if show_example_errors:
        print("Example errors:")
        plot_example_errors(cls_pred=cls_pred, correct=correct)
```

Now that we have our auxiliary functions, we can start the optimization. At the first place, let's iterate the fine-tuning 10,000 times and see the performance:

```
optimize(num_iterations=1000)
```

After 10,000 iterations, we observe the following result:

```
Accuracy on Test-Set: 78.8% (3150 / 4000)
Precision: 0.793378626929
Recall: 0.7875
F1-score: 0.786639298213
```

This means the accuracy on the test set is about 79%. Also, let's see how well our classifier performs on a sample image:

Figure 7: Random prediction on the test set (after 10,000 iterations)

After that, we further iterate the optimization up to 100,000 times and observe better accuracy:

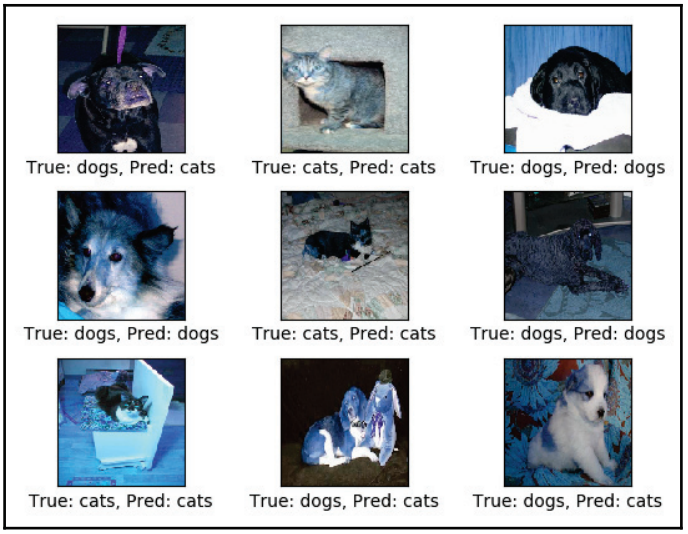


Figure 8: Random prediction on the test set (after 100,000 iterations)

```
>>>
Accuracy on Test-Set: 81.1% (3244 / 4000)
Precision: 0.811057239265
Recall: 0.811
F1-score: 0.81098298755
```

So it did not improve that much but was a 2% increase on the overall accuracy. Now is the time to evaluate our model for a single image. For simplicity, we will take two random images of a dog and a cat and see the prediction power of our model:

Figure 9: Example image for the cat and dog to be classified

At first, we load these two images and prepare the test set accordingly, as we have seen in an earlier step in this example:

```
test_cat = cv2.imread('Test_image/cat.jpg')
test_cat = cv2.resize(test_cat, (img_size, img_size), cv2.INTER_LINEAR) /
255
preview_cat = plt.imshow(test_cat.reshape(img_size, img_size,
num_channels))

test_dog = cv2.imread('Test_image/dog.jpg')
test_dog = cv2.resize(test_dog, (img_size, img_size), cv2.INTER_LINEAR) /
255
preview_dog = plt.imshow(test_dog.reshape(img_size, img_size,
num_channels))
```

Then we have the following function for making the prediction:

```
def sample_prediction(test_im):
    feed_dict_test = {
        x: test_im.reshape(1, img_size_flat),
        y_true: np.array([[1, 0]])
    }
    test_pred = session.run(y_pred_cls, feed_dict=feed_dict_test)
    return classes[test_pred[0]]
print("Predicted class for test_cat:
{}".format(sample_prediction(test_cat)))
print("Predicted class for test_dog:
{}".format(sample_prediction(test_dog)))

>>>
Predicted class for test_cat: cats
Predicted class for test_dog: dogs
```

Finally, when we're done, we close the TensorFlow session by invoking the `close()` method:

```
session.close()
```

Model performance optimization

Since CNNs are different from the layering architecture's perspective, they have different requirements as well as tuning criteria. How do you know what combination of hyperparameters is the best for your task? Of course, you can use a grid search with cross-validation to find the right hyperparameters for linear machine learning models.

However, for CNNs, there are many hyperparameters to tune, and since training a neural network on a large dataset takes a lot of time, you will only be able to explore a tiny part of the hyperparameter space in a reasonable amount of time. Here are some insights that can be followed.

Number of hidden layers

For many problems, you can just begin with a single hidden layer and you will get reasonable results. It has actually been shown that an MLP with just one hidden layer can model even the most complex functions provided it has enough neurons. For a long time, these facts convinced researchers that there was no need to investigate any deeper neural networks. However, they overlooked the fact that deep networks have a much higher parameter efficiency than shallow ones; they can model complex functions using exponentially fewer neurons than shallow nets, making them much faster to train.

It is to be noted that this might not be always the case. However, in summary, for many problems, you can start with just one or two hidden layers. It will work just fine using two hidden layers with the same total amount of neurons, in roughly the same amount of training time. For a more complex problem, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers and a huge amount of training data.

Number of neurons per hidden layer

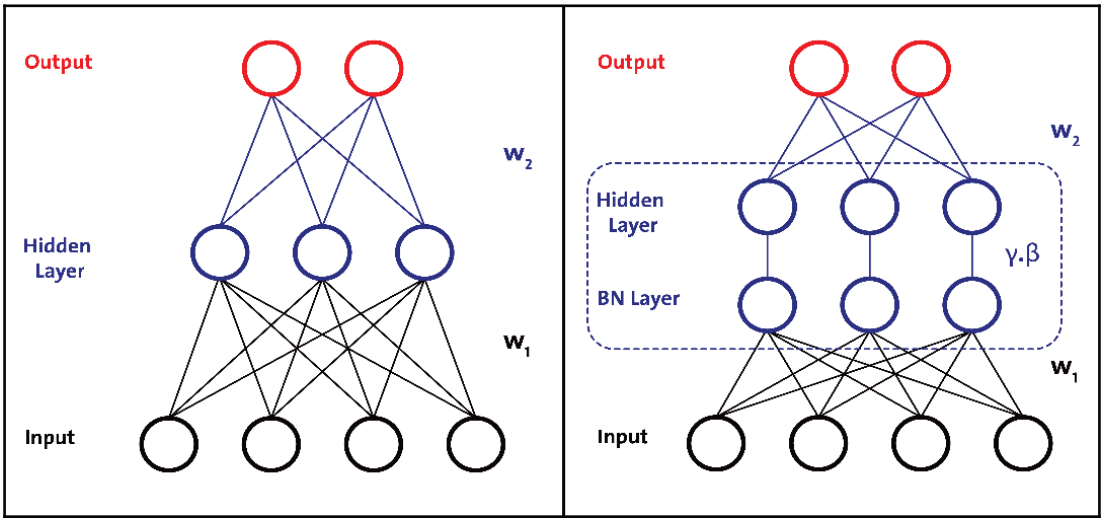
Obviously, the number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, if your dataset has the shape of 28 x 28 it should expect to have input neurons with size 784 and the output neurons should be equal to the number of classes to be predicted. As for the hidden layers, a common practice is to size them to form a funnel, with fewer and fewer neurons at each layer, the rationale being that many low-level features can coalesce into far fewer high-level features. However, this practice is not as common now, and you may simply use the same size for all hidden layers.

If there are four convolutional layers with 256 neurons, that's just one hyperparameter to tune instead of one per layer. Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. Another important question is: when would you want to add a max pooling layer rather than a convolutional layer with the same stride? The thing is that a max-pooling layer has no parameters at all, whereas a convolutional layer has quite a few.

Sometimes, adding a local response normalization layer that makes the neurons that most strongly activate inhibit neurons at the same location but in neighboring feature maps, encourages different feature maps to specialize and pushes them apart, forcing them to explore a wider range of features. It is typically used in the lower layers to have a larger pool of low-level features that the upper layers can build upon.

Batch normalization

Batch normalization (**BN**) is a method to reduce internal covariate shift while training regular DNNs. This can apply to CNNs too. Due to the normalization, BN further prevents smaller changes to the parameters to amplify and thereby allows higher learning rates, making the network even faster:



The idea is placing an additional step between the layers, in which the output of the layer before is normalized. To be more specific, in the case of non-linear operations (for example, ReLU), BN transformation has to be applied to the non-linear operation. Typically, the overall process has the following workflow:

- Transforming the network into a BN network (see *Figure 1*)
- Then training the new network
- Transforming the batch statistic into a population statistic

This way, BN can fully partake in the process of backpropagation. As shown in *Figure 1*, BN is performed before the other processes of the network in this layer are applied. However, any kind of gradient descent (for example, **stochastic gradient descent** (**SGD**) and its variants) can be applied to train the BN network.

Interested readers can refer to the original paper to get to more information: Ioffe, Sergey, and Christian Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift*. arXiv preprint arXiv:1502.03167 (2015).

Now a valid question would be: where to place the BN layer? Well, to know the answer, a quick evaluation of BatchNorm layer performance on ImageNet-2012 (`https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md`) shows the following benchmark:

Name	Accuracy	LogLoss	Comments
Before	0.474	2.35	As in paper
Before + scale&bias layer	0.478	2.33	As in paper
After	**0.499**	**2.21**	
After + scale&bias layer	0.493	2.24	

From the preceding table, it can be seen that placing BN after non-linearity would be the right way. The second question would be: what activation function should be used in a BN layer? Well, from the same benchmark, we can see the following result:

Name	Accuracy	LogLoss	Comments
ReLU	0.499	2.21	
RReLU	0.500	2.20	
PReLU	**0.503**	**2.19**	
ELU	0.498	2.23	
Maxout	0.487	2.28	
Sigmoid	0.475	2.35	
TanH	0.448	2.50	
No	0.384	2.96	

From the preceding table, we can assume that using ReLU or its variants would be a better idea. Now, another question would be how to use these using deep learning libraries. Well, in TensorFlow, it is:

```
training = tf.placeholder(tf.bool)
x = tf.layers.dense(input_x, units=100)
x = tf.layers.batch_normalization(x, training=training)
x = tf.nn.relu(x)
```

A general warning: set this to `True` for training and `False` for testing. However, the preceding addition introduces extra ops to be performed on the graph, which is updating its mean and variance variables in such a way that they will not be dependencies of your training op. To do it, we can just run the ops separately, as follows:

```
extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
sess.run([train_op, extra_update_ops], ...)
```

Advanced regularization and avoiding overfitting

As mentioned in the previous chapter, one of the main disadvantages observed during the training of large neural networks is overfitting, that is, generating very good approximations for the training data but emitting noise for the zones between single points. There are a couple of ways to reduce or even prevent this issue, such as dropout, early stop, and limiting the number of parameters.

In the case of overfitting, the model is specifically adjusted to the training dataset, so it will not be used for generalization. Therefore, although it performs well on the training set, its performance on the test dataset and subsequent tests is poor because it lacks the generalization property:

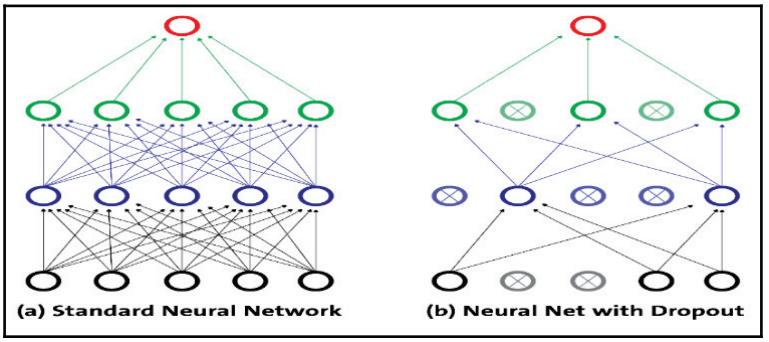


Figure 10: Dropout versus without dropout

The main advantage of this method is that it avoids holding all the neurons in a layer to optimize their weights synchronously. This adaptation made in random groups prevents all the neurons from converging to the same goals, thus de-correlating the adapted weights. A second property found in the dropout application is that the activation of the hidden units becomes sparse, which is also a desirable characteristic.

In the preceding figure, we have a representation of an original fully connected multilayer neural network and the associated network with the dropout linked. As a result, approximately half of the input was zeroed (this example was chosen to show that probabilities will not always give the expected four zeroes). One factor that could have surprised you is the scale factor applied to the non-dropped elements.

This technique is used to maintain the same network, and restore it to the original architecture when training, using `dropout_keep_prob` as 1. A major drawback of using dropout is that it does not have the same benefits for convolutional layers, where the neurons are not fully connected. To address this issue, there are a few techniques can be applied, such as DropConnect and stochastic pooling:

- DropConnect is similar to dropout as it introduces dynamic sparsity within the model, but it differs in that the sparsity is on the weights, rather than the output vectors of a layer. The thing is that a fully connected layer with DropConnect becomes a sparsely connected layer in which the connections are chosen at random during the training stage.
- In stochastic pooling, the conventional deterministic pooling operations are replaced with a stochastic procedure, where the activation within each pooling region is picked randomly according to a multinomial distribution, given by the activities within the pooling region. The approach is hyperparameter free and can be combined with other regularization approaches, such as dropout and data augmentation.

Stochastic pooling versus standard max pooling: Stochastic pooling is equivalent to standard max pooling but with many copies of an input image, each having small local deformations.

Secondly, one of the simplest methods to prevent overfitting of a network is to simply stop the training before overfitting gets a chance to occur. This comes with the disadvantage that the learning process is halted. Thirdly, limiting the number of parameters is sometimes helpful and helps avoid overfitting. When it comes to CNN training, the filter size also affects the number of parameters. Thus, limiting this type of parameter restricts the predictive power of the network directly, reducing the complexity of the function that it can perform on the data, and that limits the amount of overfitting.

Applying dropout operations with TensorFlow

If we apply the dropout operation to a sample vector, it will work on transmitting the dropout to all the architecture-dependent units. In order to apply the dropout operation, TensorFlow implements the `tf.nn.dropout` method, which works as follows:

```
tf.nn.dropout (x, keep_prob, noise_shape, seed, name)
```

Where `x` is the original tensor. The `keep_prob` means the probability of keeping a neuron and the factor by which the remaining nodes are multiplied. The `noise_shape` signifies a four-element list that determines whether a dimension will apply zeroing independently or not. Let's have a look at this code segment:

```
import tensorflow as tf X = [1.5, 0.5, 0.75, 1.0, 0.75, 0.6, 0.4, 0.9]
drop_out = tf.nn.dropout(X, 0.5)
sess = tf.Session() with sess.as_default():
    print(drop_out.eval())
sess.close()

[ 3. 0. 1.5 0. 0. 1.20000005 0. 1.79999995]
```

In the preceding example, you can see the results of applying dropout to the *x* variable, with a 0.5 probability of zero; in the cases in which it didn't occur, the values were doubled (multiplied by 1/1.5, the dropout probability).

Which optimizer to use?

When using a CNN, since one of the objective functions is to minimize the evaluated cost, we must define an optimizer. Using the most common optimizer , such as SGD, the learning rates must scale with *1/T* to get convergence, where *T* is the number of iterations. Adam or RMSProp try to overcome this limitation automatically by adjusting the step size so that the step is on the same scale as the gradients. In addition, in the previous example, we have used Adam optimizer, which performs well in most cases.

Nevertheless, if you are training a neural network but computing the gradients is mandatory, using the `RMSPropOptimizer` function (which implements the `RMSProp` algorithm) is a better idea since it would be the faster way of learning in a mini-batch setting. Researchers also recommend using the momentum optimizer, while training a deep CNN or DNN. Technically, `RMSPropOptimizer` is an advanced form of gradient descent that divides the learning rate by an exponentially decaying average of squared gradients. The suggested setting value of the decay parameter is 0.9, while a good default value for the learning rate is 0.001. For example, in TensorFlow, `tf.train.RMSPropOptimizer()` helps us to use this with ease:

```
optimizer = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost_op)
```

Memory tuning

In this section, we try to provide some insights. We start with an issue and its solution; convolutional layers require a huge amount of RAM, especially during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass. During inference (that is, when making a prediction for a new instance), the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers.

Nevertheless, during training, everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers. If your GPU runs out of memory while training a CNN, here are five things you can try to solve the problem (other than purchasing a GPU with more RAM):

- Reduce the mini-batch size
- Reduce dimensionality using a larger stride in one or more layers
- Remove one or more layers
- Use 16-bit floats instead of 32-bit
- Distribute the CNN across multiple devices (see more at `https://www.tensorflow.org/deploy/distributed`)

Appropriate layer placement

Another important question would be: when do you want to add a max pooling layer rather than a convolutional layer with the same stride? The thing is that a max-pooling layer has no parameters at all, whereas a convolutional layer has quite a few.

Even adding a local response normalization layer sometimes makes the neurons that most strongly activate inhibit neurons at the same location but in neighboring feature maps, which encourages different feature maps to specialize and pushes them apart, forcing them to explore a wider range of features. It is typically used in the lower layers to have a larger pool of low-level features that the upper layers can build upon.

Building the second CNN by putting everything together

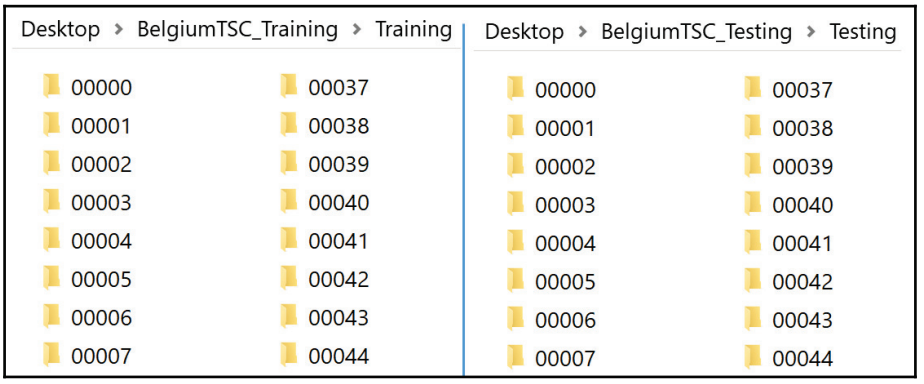
Now we know how to optimize the layering structure in a CNN by adding dropout, BN, and biases initializers, such as Xavier. Let's try to apply these to a less complex CNN. Throughout this example, we will see how to solve a real-life classification problem. To be more specific, our CNN model will be able to classify the traffic sign from a bunch of images.

Dataset description and preprocessing

For this we will be using the Belgian traffic dataset (BelgiumTS for Classification (cropped images)). This dataset can be download from `http://btsd.ethz.ch/shareddata/`. Here are a quick glimpse about the traffic signs convention in Belgium:

- Belgian traffic signs are usually in Dutch and French. This is good to know, but for the dataset that you'll be working with, it's not too important!
- There are six categories of traffic signs in Belgium: warning signs, priority signs, prohibitory signs, mandatory signs, signs related to parking and standing still on the road and, lastly, designatory signs.

Once we download the aforementioned dataset, we will see the following directory structure (training left, test right):



The images are in `.ppm` format; otherwise we could've used TensorFlow built-in image loader (example, `tf.image.decode_png`). However, we can use the `skimage` Python package.

In Python 3, execute `$ sudo pip3 install scikit-image` for `skimage` to install and use this package. So let's get started by showing the directory path as follows:

```
Train_IMAGE_DIR = "<path>/BelgiumTSC_Training/"
Test_IMAGE_DIR = ""<path>/BelgiumTSC_Testing/"
```

Then let's write a function using the `skimage` library to read the images and returns two lists:

- `images`: A list of Numpy arrays, each representing an image
- `labels`: A list of numbers that represent the images labels

```
def load_data(data_dir):
    # All subdirectories, where each folder represents a unique label
    directories = [d for d in os.listdir(data_dir)if
os.path.isdir(os.path.join(data_dir, d))]

    # Iterate label directories and collect data in two lists, labels and
images.
    labels = []
    images = []
    for d in directories:label_dir = os.path.join(data_dir, d)
    file_names = [os.path.join(label_dir, f)
                for f in os.listdir(label_dir) if f.endswith(".ppm")]

    # For each label, load it's images and add them to the images list.
    # And add the label number (i.e. directory name) to the labels list.
    for f in file_names:images.append(skimage.data.imread(f))
    labels.append(int(d))
    return images, labels
```

The preceding code block is straightforward and contains inline comments. How about showing related statistics about images? However, before that, let's invoke the preceding function:

```
# Load training and testing datasets.
train_data_dir = os.path.join(Train_IMAGE_DIR, "Training")
test_data_dir = os.path.join(Test_IMAGE_DIR, "Testing")

images, labels = load_data(train_data_dir)
```

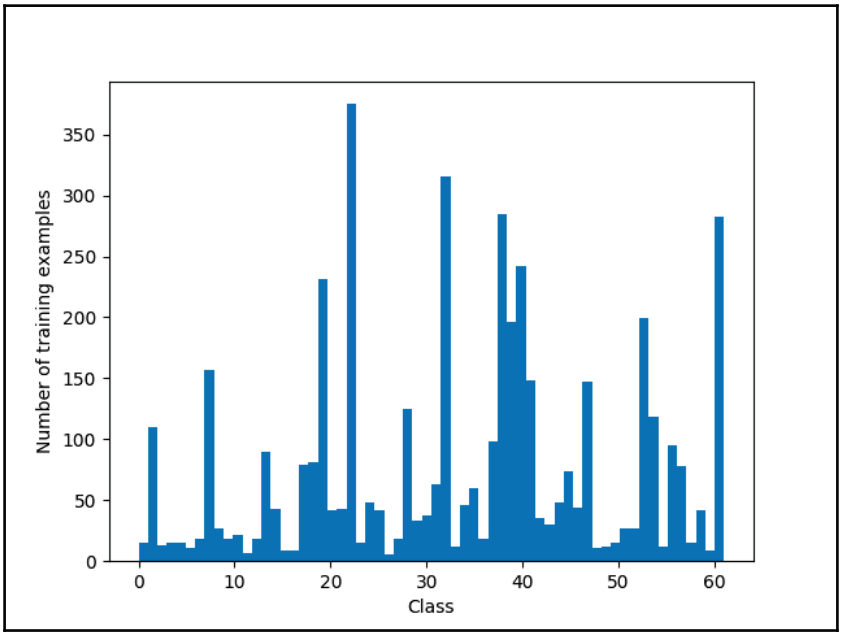
Then let's see some statistics:

```
print("Unique classes: {0} \nTotal Images: {1}".format(len(set(labels)),
len(images)))

>>>
Unique classes: 62
Total Images: 4575
```

So we have 62 classes to be predicted (that is, a multiclass image classification problem) and we have many images too that should be sufficient to satisfy a smaller CNN.Now let's see the class distribution visually:

```
# Make a histogram with 62 bins of the `labels` data and show the plot:
plt.hist(labels, 62)
plt.xlabel('Class')
plt.ylabel('Number of training examples')
plt.show()
```



Therefore, from the preceding figure, we can see that classes are very imbalanced. However, to make it simpler, we won't take care of this but next, it would be great to visually inspect some files, say displaying the first image of each label:

```
def display_images_and_labels(images, labels):
    unique_labels = set(labels)
    plt.figure(figsize=(15, 15))
    i = 1
    for label in unique_labels:
        # Pick the first image for each label.
        image = images[labels.index(label)]
        plt.subplot(8, 8, i) # A grid of 8 rows x 8 column
        splt.axis('off')
        plt.title("Label {0} ({1})".format(label, labels.count(label)))
        i += 1
        _= plt.imshow(image)
        plt.show()
display_images_and_labels(images, labels)
```

Now you can see from the preceding figure that the images come in different sizes and shapes. Moreover, we can see it using Python code, as follows:

```
for img in images[:5]:
    print("shape: {0}, min: {1}, max: {2}".format(img.shape, img.min(),
img.max()))

>>>
shape: (87, 84, 3), min: 12, max: 255
shape: (289, 169, 3), min: 0, max: 255
shape: (205, 76, 3), min: 0, max: 255
shape: (72, 71, 3), min: 14, max: 185
shape: (231, 228, 3), min: 0, max: 255
```

Therefore, we need to apply some pre-processing such as resizing, reshaping, and so on to each image. Let's say each image will have size of 32 x 32:

```
images32 = [skimage.transform.resize(img, (32, 32), mode='constant')

for img in images]for img in images32[:5]:
    print("shape: {0}, min: {1}, max: {2}".format(img.shape, img.min(),
img.max()))

>>>
shape: (32, 32, 3), min: 0.06642539828431372, max: 0.9704350490196079
shape: (32, 32, 3), min: 0.0, max: 1.0
shape: (32, 32, 3), min: 0.03172870710784261, max: 1.0
shape: (32, 32, 3), min: 0.059474571078431314, max: 0.7036305147058846
shape: (32, 32, 3), min: 0.01506204044117481, max: 1.0
```

Now, all of our images have same size. The next task would be to convert labels and image features as a `numpy` array:

```
labels_array = np.array(labels)
images_array = np.array(images32)
print("labels: ", labels_array.shape, "nimages: ", images_array.shape)

>>>
labels: (4575,)
images: (4575, 32, 32, 3)
```

Fantastic! The next task would be creating our second CNN, but this time we will be using TensorFlow `contrib` package, which is a high-level API that supports layering ops.

Creating the CNN model

We are going to construct a complex network. However, it has a straightforward architecture. At the beginning, we use Xavier as the network initializer. Once we initialize the network bias using the Xavier initializer. The input layer is followed by a convolutional layer (convolutional layer 1), which is again followed by a BN layer (that is, BN layer 1). Then there is a pooling layer with strides of two and a kernel size of two. Then another BN layer follows the second convolutional layer. Next, there is the second pooling layer with strides of two and kernel size of two. Well, then the max polling layer is followed by a flattening layer that flattens the input from (None, height, width, channels) to (None, height * width * channels) == (None, 3072).

Once the flattening is completed, the input is fed into the first fully connected layer 1. Then third BN is applied as a normalizer function. Then we will have a dropout layer before we feed the lighter network into the fully connected layer 2 that generates logits of size (None, 62). Too much of a mouthful? Don't worry; we will see it step by step. Let's start the coding by creating the computational graph, creating both features, and labeling placeholders:

```
graph = tf.Graph()
with graph.as_default():
    # Placeholders for inputs and labels.
    images_X = tf.placeholder(tf.float32, [None, 32, 32, 3]) # each image's
32x32 size
    labels_X = tf.placeholder(tf.int32, [None])

    # Initializer: Xavier
     biasInit = tf.contrib.layers.xavier_initializer(uniform=True,
seed=None, dtype=tf.float32)

    # Convolution layer 1: number of neurons 128 and kernel size is 6x6.
     conv1 = tf.contrib.layers.conv2d(images_X, num_outputs=128,
kernel_size=[6, 6],
            biases_initializer=biasInit)

    # Batch normalization layer 1: can be applied as a normalizer
    # function for conv2d and fully_connected
    bn1 = tf.contrib.layers.batch_norm(conv1, center=True, scale=True,
is_training=True)

    # Max Pooling (down sampling) with strides of 2 and kernel size of 2
    pool1 = tf.contrib.layers.max_pool2d(bn1, 2, 2)

    # Convolution layer 2: number of neurons 256 and kernel size is 6x6.
    conv2 = tf.contrib.layers.conv2d(pool1, num_outputs=256,
kernel_size=[4, 4], stride=2,
```

```
        biases_initializer=biasInit)

    # Batch normalization layer 2:
    bn2 = tf.contrib.layers.batch_norm(conv2, center=True, scale=True,
is_training=True)

    # Max Pooling (down-sampling) with strides of 2 and kernel size of 2
    pool2 = tf.contrib.layers.max_pool2d(bn2, 2, 2)

    # Flatten the input from [None, height, width, channels] to
    # [None, height * width * channels] == [None, 3072]
    images_flat = tf.contrib.layers.flatten(pool2)

    # Fully connected layer 1
    fc1 = tf.contrib.layers.fully_connected(images_flat, 512, tf.nn.relu)

    # Batch normalization layer 3
    bn3 = tf.contrib.layers.batch_norm(fc1, center=True, scale=True,
is_training=True)

    # apply dropout, if is_training is False, dropout is not applied
    fc1 = tf.layers.dropout(bn3, rate=0.25, training=True)

    # Fully connected layer 2 that generates logits of size [None, 62].
    # Here 62 means number of classes to be predicted.
    logits = tf.contrib.layers.fully_connected(fc1, 62, tf.nn.relu)
```

Up to this point, we have managed to generate the logits of size (`None, 62`). Then we need to convert the logits to label indexes (`int`) with the shape (`None`), which is a 1D vector of `length == batch_size:predicted_labels = tf.argmax(logits, axis=1)`. Then we define cross-entropy as the `loss` function, which is a good choice for classification:

```
loss_op =
tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits
, labels=labels_X))
```

Now one of the most important parts is updating the ops and creating an optimizer (Adam in our case):

```
update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.control_dependencies(update_ops):
    # Create an optimizer, which acts as the training op.train =
    tf.train.AdamOptimizer(learning_rate=0.10).minimize(loss_op)
```

Finally, we initialize all the ops:

```
init_op = tf.global_variables_initializer()
```

Training and evaluating the network

We start by create a session to run the graph we created. Note that for faster training, we should use a GPU. However, if you do not have a GPU, just set `log_device_placement=False`:

```
session = tf.Session(graph=graph,
config=tf.ConfigProto(log_device_placement=True))
session.run(init_op)
for i in range(300):
    _, loss_value = session.run([train, loss_op], feed_dict={images_X:
images_array, labels_X:
    labels_array})
    if i % 10 == 0:
        print("Loss: ", loss_value)

>>>
Loss: 4.7910895
Loss: 4.3410876
Loss: 4.0275432
...
Loss: 0.523456
```

Once the training is completed, let us pick 10 random images and see the predictive power of our model:

```
random_indexes = random.sample(range(len(images32)), 10)
random_images = [images32[i]
for i in random_indexes]
    random_labels = [labels[i]
for i in random_indexes]
```

Then let's run the `predicted_labels op`:

```
predicted = session.run([predicted_labels], feed_dict={images_X:
random_images})[0]
print(random_labels)
print(predicted)

>>>
[38, 21, 19, 39, 22, 22, 45, 18, 22, 53]
[20  21  19  51  22  22  45  53  22  53]
```

So we can see that some images were correctly classified and some wrongly. However, visual inspection would be more helpful. So let's display the predictions and the ground truth:

```
fig = plt.figure(figsize=(5, 5))
for i in range(len(random_images)):
    truth = random_labels[i]
    prediction = predicted[i]
    plt.subplot(5, 2,1+i)
    plt.axis('off')color='green'
    if truth == prediction
    else
     'red'plt.text(40, 10, "Truth: {0}nPrediction: {1}".format(truth,
prediction), fontsize=12,
    color=color)
plt.imshow(random_images[i])
>>>
```

Finally, we can evaluate our model using the test set. To see the predictive power, we compute the accuracy:

```
# Load the test dataset.
test_X, test_y = load_data(test_data_dir)

# Transform the images, just as we did with the training set.
test_images32 = [skimage.transform.resize(img, (32, 32), mode='constant')
for img in test_X]
```

```
    display_images_and_labels(test_images32, test_y)

# Run predictions against the test
setpredicted = session.run([predicted_labels], feed_dict={images_X:
test_images32})[0]

# Calculate how many matches
match_count = sum([int(y == y_) for y, y_ in zip(test_y, predicted)])
accuracy = match_count / len(test_y)print("Accuracy:
{:.3f}".format(accuracy))

>>
Accuracy: 87.583
```

Not that bad in terms of accuracy. In addition to this, we can also compute other performance metrics such as precision, recall, f1 measure and also visualize the result in a confusion matrix to show the predicted versus actual labels count. Nevertheless, we can still improve the accuracy by tuning the network and hyperparameters. But I leave these up to the readers.

Finally, we are done, so let's close the TensorFlow session:

```
session.close()
```

Summary

In this chapter, we discussed how to use CNNs, which are a type of feed-forward artificial neural network in which the connectivity pattern between neurons is inspired by the organization of an animal's visual cortex. We saw how to cascade a set of layers to construct a CNN and perform different operations in each layer. Then we saw how to train a CNN. Later on, we discussed how to optimize the CNN hyperparameters and optimization.

Finally, we built another CNN, where we utilized all the optimization techniques. Our CNN models did not achieve outstanding accuracy since we iterated both of the CNNs a few times and did not even apply any grid searching techniques; that means we did not hunt for the best combinations of the hyperparameters. Therefore, the takeaway would be to apply more robust feature engineering in the raw images, iterate the training for more epochs with the best hyperparameters, and observe the performance.

In the next chapter, we will see how to use some deeper and popular CNN architectures, such as ImageNet, AlexNet, VGG, GoogLeNet, and ResNet. We will see how to utilize these trained models for transfer learning.

4

Popular CNN Model Architectures

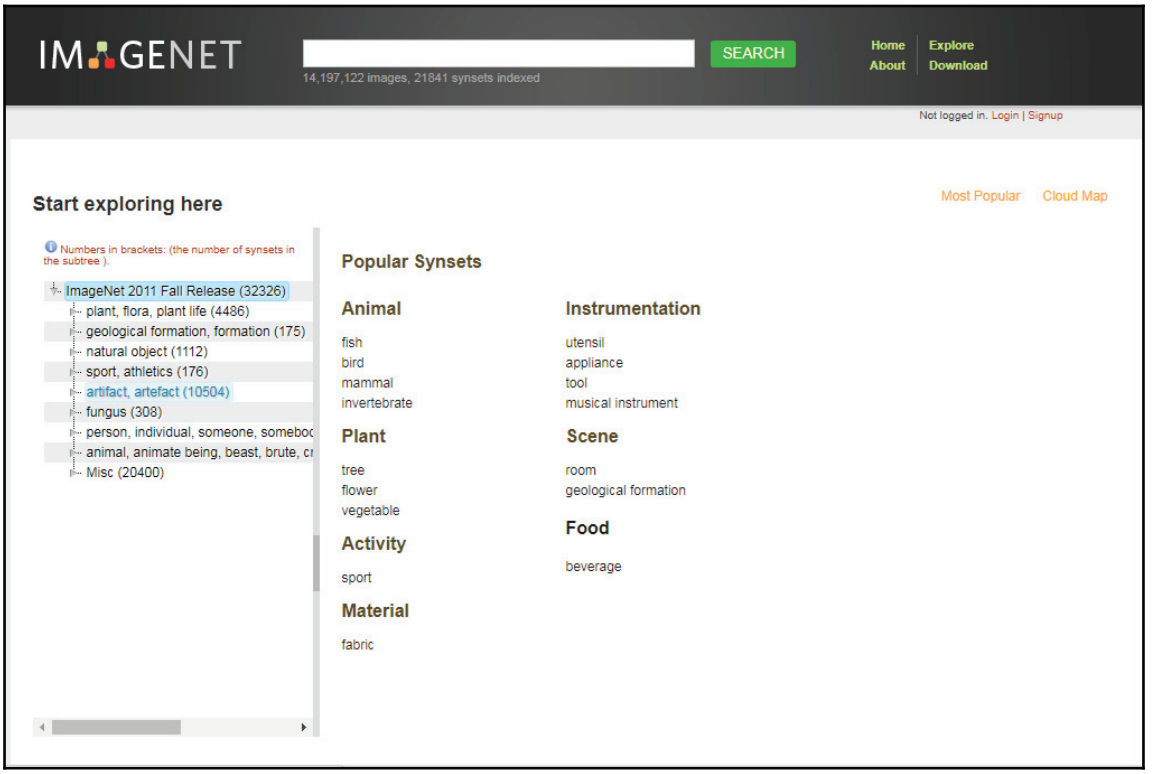
In this chapter, will introduce the ImageNet image database and also cover the architectures of the following popular CNN models:

- LeNet
- AlexNet
- VGG
- GoogLeNet
- ResNet

Introduction to ImageNet

ImageNet is a database of over 15 million hand-labeled, high-resolution images in roughly 22,000 categories. This database is organized just like the WordNet hierarchy, where each concept is also called a **synset** (that is, **synonym set**). Each synset is a node in the ImageNet hierarchy. Each node has more than 500 images.

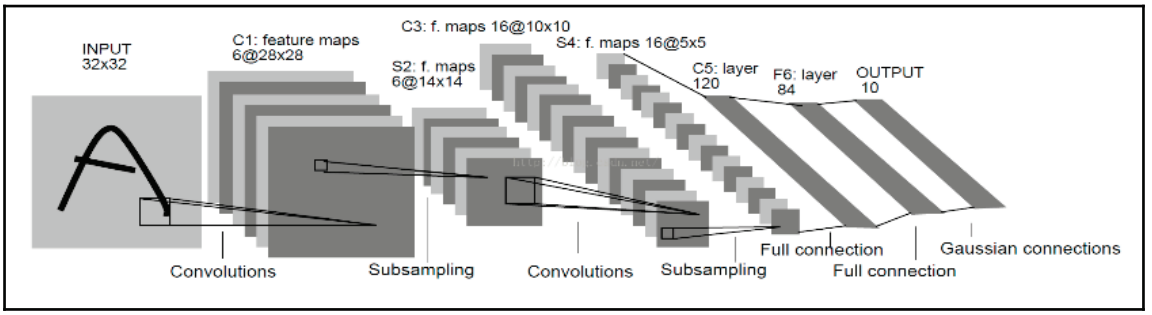
The **ImageNet Large Scale Visual Recognition Challenge** (**ILSVRC**) was founded in 2010 to improve state-of-the-art technology for object detection and image classification on a large scale:



Following this overview of ImageNet, we will now take a look at various CNN model architectures.

LeNet

In 2010, a challenge from ImageNet (known as **ILSVRC 2010**) came out with a CNN architecture, LeNet 5, built by Yann Lecun. This network takes a 32 x 32 image as input, which goes to the convolution layers (**C1**) and then to the subsampling layer (**S2**). Today, the subsampling layer is replaced by a pooling layer. Then, there is another sequence of convolution layers (**C3**) followed by a pooling (that is, subsampling) layer (**S4**). Finally, there are three fully connected layers, including the **OUTPUT** layer at the end. This network was used for zip code recognition in post offices. Since then, every year various CNN architectures were introduced with the help of this competition:



LeNet 5 – CNN architecture from Yann Lecun's article in 1998

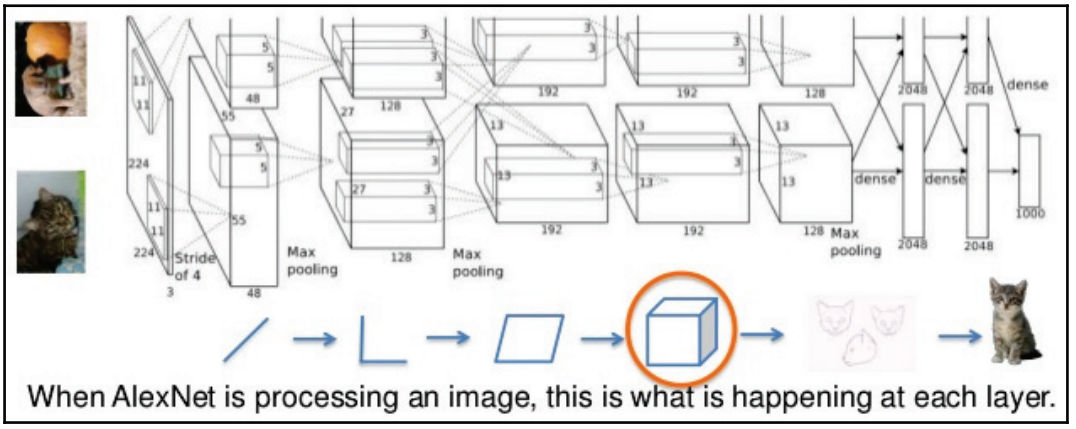
Therefore, we can conclude the following points:

- The input to this network is a grayscale 32 x 32 image
- The architecture implemented is a CONV layer, followed by POOL and a fully connected layer
- CONV filters are 5 x 5, applied at a stride of 1

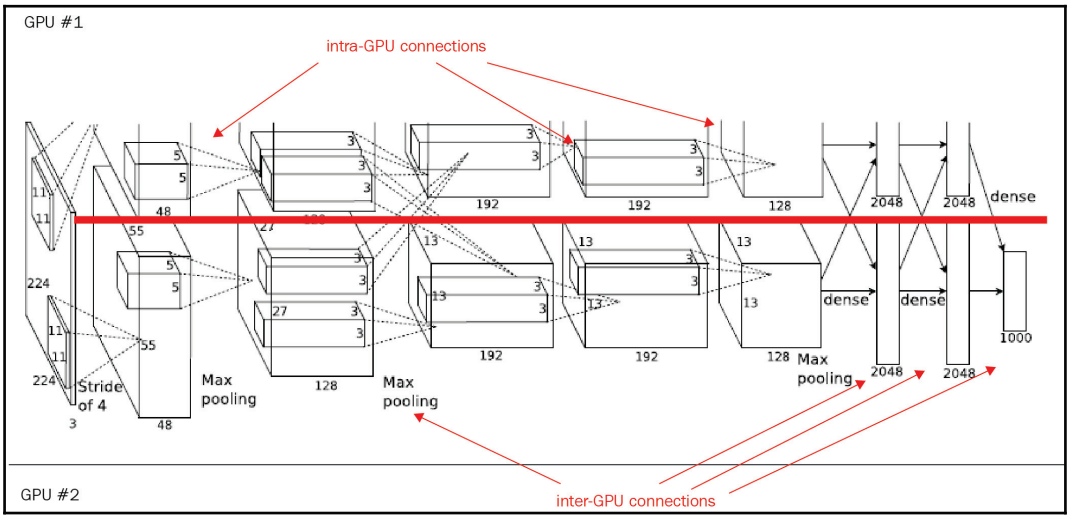
AlexNet architecture

The first breakthrough in the architecture of CNN came in the year 2012. This award-winning CNN architecture is called **AlexNet**. It was developed at the University of Toronto by Alex Krizhevsky and his professor, Jeffry Hinton.

In the first run, a ReLU activation function and a dropout of 0.5 were used in this network to fight overfitting. As we can see in the following image, there is a normalization layer used in the architecture, but this is not used in practice anymore as it used heavy data augmentation. AlexNet is still used today even though there are more accurate networks available, because of its relative simple structure and small depth. It is widely used in computer vision:



AlexNet is trained on the ImageNet database using two separate GPUs, possibly due to processing limitations with inter-GPU connections at the time, as shown in the following figure:



Traffic sign classifiers using AlexNet

In this example, we will use transfer learning for feature extraction and a German traffic sign dataset to develop a classifier. Used here is an AlexNet implementation by Michael Guerzhoy and Davi Frossard, and AlexNet weights are from the Berkeley vision and Learning center. The complete code and dataset can be downloaded from here.

AlexNet expects a 227 x 227 x 3 pixel image, whereas the traffic sign images are 32 x 32 x 3 pixels. In order to feed the traffic sign images into AlexNet, we'll need to resize the images to the dimensions that AlexNet expects, that is, 227 x 227 x 3:

```
original_image = tf.placeholder(tf.float32, (None, 32, 32, 3))
resized_image = tf.image.resize_images(original_imag, (227, 227))
```

We can do so with the help of the `tf.image.resize_images` method by TensorFlow. Another issue here is that AlexNet was trained on the ImageNet dataset, which has 1,000 classes of images. So, we will replace this layer with a 43-neuron classification layer. To do this, figure out the size of the output from the last fully connected layer; since this is a fully connected layer and so is a 2D shape, the last element will be the size of the output. `fc7.get_shape().as_list()[-1]` does the trick; combine this with the number of classes for the traffic sign dataset to get the shape of the final fully connected layer: `shape = (fc7.get_shape().as_list()[-1], 43)`. The rest of the code is just the standard way to define a fully connected layer in TensorFlow. Finally, calculate the probabilities with `softmax`:

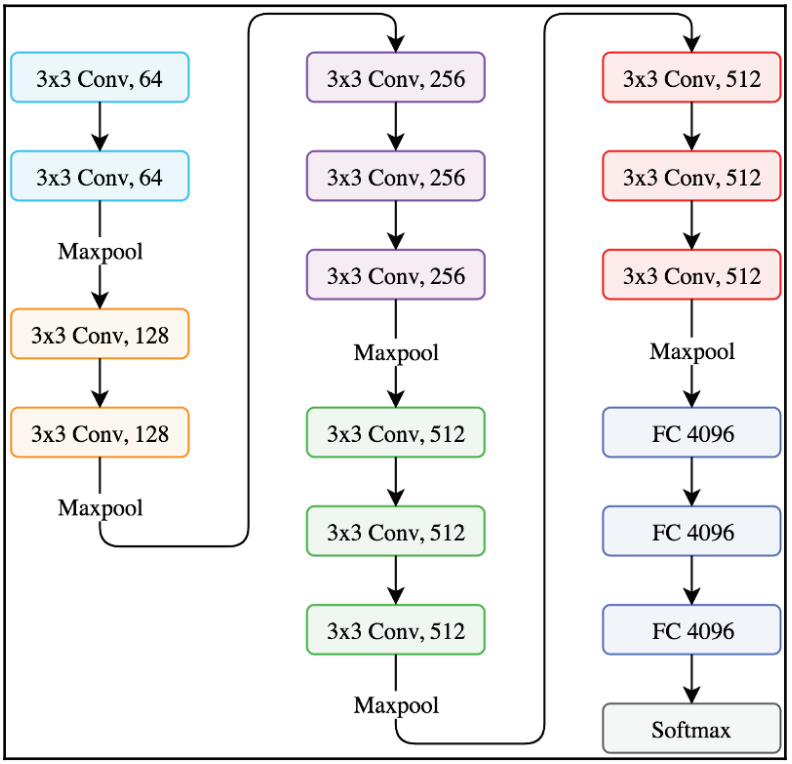
```
#Refer AlexNet implementation code, returns last fully connected layer
fc7 = AlexNet(resized, feature_extract=True)
shape = (fc7.get_shape().as_list()[-1], 43)
fc8_weight = tf.Variable(tf.truncated_normal(shape, stddev=1e-2))
fc8_b = tf.Variable(tf.zeros(43))
logits = tf.nn.xw_plus_b(fc7, fc8_weight, fc8_b)
probs = tf.nn.softmax(logits)
```

VGGNet architecture

The runner-up in the 2014 ImageNet challenge was VGGNet from the visual geometric group at Oxford University. This convolutional neural network is a simple and elegant architecture with a 7.3% error rate. It has two versions: VGG16 and VGG19.

VGG16 is a 16-layer neural network, not counting the max pooling layer and the softmax layer. Hence, it is known as VGG16. VGG19 consists of 19 layers. A pre-trained model is available in Keras for both Theano and TensorFlow backends.

The key design consideration here is depth. Increases in the depth of the network were achieved by adding more convolution layers, and it was done due to the small 3 x 3 convolution filters in all the layers. The default input size of an image for this model is 224 x 224 x 3. The image is passed through a stack of convolution layers with a stride of 1 pixel and padding of 1. It uses 3 x 3 convolution throughout the network. Max pooling is done over a 2 x 2 pixel window with a stride of 2, then another stack of convolution layers followed by three fully connected layers. The first two fully connected layers have 4,096 neurons each, and the third fully connected layers are responsible for classification with 1,000 neurons. The final layer is a softmax layer. VGG16 uses a much smaller 3 x 3 convolution window, compared to AlexNet's much larger 11 x 11 convolution window. All hidden layers are built with the ReLU activation function. The architecture looks like this:



VGG16 network architecture

Due to the small 3 x 3 convolution filter, the depth of VGGNet is increased. The number of parameters in this network is approximately 140 million, mostly from the first fully connected layer. In latter-day architectures, fully connected layers of VGGNet are replaced with **global average pooling** (**GAP**) layers in order to minimize the number of parameters.

Another observation is that the number of filters increases as the image size decreases.

VGG16 image classification code example

The Keras Applications module has pre-trained neural network models, along with its pre-trained weights trained on ImageNet. These models can be used directly for prediction, feature extraction, and fine-tuning:

```
#import VGG16 network model and other necessary libraries

from keras.applications.vgg16 import VGG16
from keras.preprocessing import image
from keras.applications.vgg16 import preprocess_input
import numpy as np

#Instantiate VGG16 and returns a vgg16 model instance
vgg16_model = VGG16(weights='imagenet', include_top=False)
#include_top: whether to include the 3 fully-connected layers at the top of
the network.
#This has to be True for classification and False for feature extraction.
Returns a model instance
#weights:'imagenet' means model is pre-training on ImageNet data.
model = VGG16(weights='imagenet', include_top=True)
model.summary()

#image file name to classify
image_path = 'jumping_dolphin.jpg'
#load the input image with keras helper utilities and resize the image.
#Default input size for this model is 224x224 pixels.
img = image.load_img(image_path, target_size=(224, 224))
#convert PIL (Python Image Library??) image to numpy array
x = image.img_to_array(img)
print (x.shape)

#image is now represented by a NumPy array of shape (224, 224, 3),
# but we need to expand the dimensions to be (1, 224, 224, 3) so we can
# pass it through the network -- we'll also preprocess the image by
# subtracting the mean RGB pixel intensity from the ImageNet dataset
```

```
#Finally, we can load our Keras network and classify the image:

x = np.expand_dims(x, axis=0)
print (x.shape)

preprocessed_image = preprocess_input(x)

preds = model.predict(preprocessed_image)
print('Prediction:', decode_predictions(preds, top=2)[0])
```

The first time it executes the preceding script, Keras will automatically download and cache the architecture weights to disk in the `~/.keras/models` directory. Subsequent runs will be faster.

GoogLeNet architecture

In 2014, ILSVRC, Google published its own network known as **GoogLeNet**. Its performance is a little better than VGGNet; GoogLeNet's performance is 6.7% compared to VGGNet's performance of 7.3%. The main attractive feature of GoogLeNet is that it runs very fast due to the introduction of a new concept called **inception module**, thus reducing the number of parameters to only 5 million; that's 12 times less than AlexNet. It has lower memory use and lower power use too.

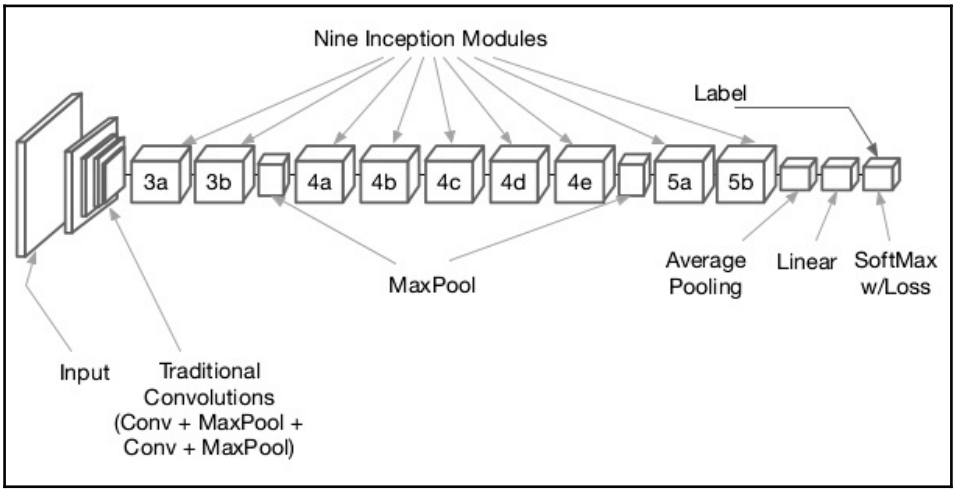
It has 22 layers, so it is a very deep network. Adding more layers increases the number of parameters and it is likely that the network overfits. There will be more computation, because a linear increase in filters results in a quadratic increase in computation. So, the designers use the inception module and GAP. The fully connected layer at the end of the network is replaced with a GAP layer because fully connected layers are generally prone to overfitting. GAP has no parameters to learn or optimize.

Architecture insights

Instead of choosing a particular filter size as in the previous architectures, the GoogLeNet designers applied all the three filters of sizes 1 x 1, 3 x 3, and 5 x 5 on the same patch, with a 3 x 3 max pooling and concatenation into a single output vector.

The use of 1 x 1 convolutions decreases the dimensions wherever the computation is increased by the expensive 3 x 3 and 5 x 5 convolutions. 1 x 1 convolutions with the ReLU activation function are used before the expensive 3 x 3 and 5 x 5 convolutions.

In GoogLeNet, inception modules are stacked one over the other. This stacking allows us to modify each module without affecting the later layers. For example, you can increase or decrease the width of any layer:



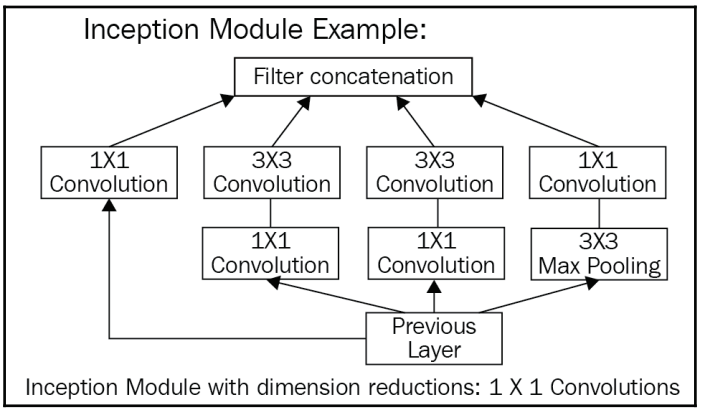
GoogLeNet architecture

Deep networks also suffer from the fear of what is known as the **vanishing gradient** problem during backpropagation. This is avoided by adding auxiliary classifiers to intermediate layers. Also, during training, the intermediate loss was added to the total loss with a discounted factor of 0.3.

Since fully connected layers are prone to overfitting, it is replaced with a GAP layer. Average pooling does not exclude use of dropout, a regularization method for overcoming overfitting in deep neural networks. GoogLeNet added a linear layer after 60, a GAP layer to help others swipe for their own classifier using transfer learning techniques.

Inception module

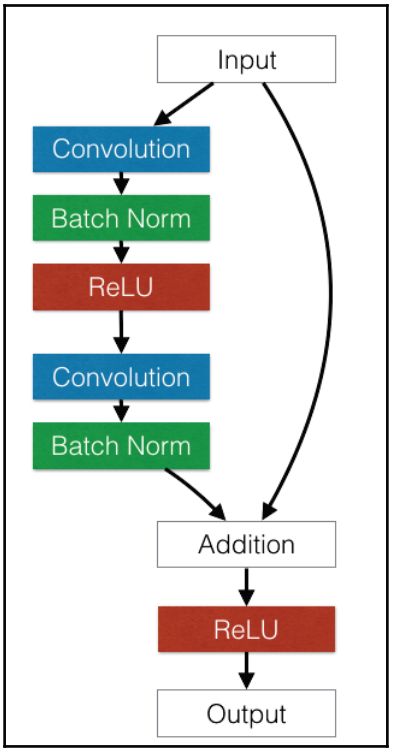
The following image is an example of an inception module:



ResNet architecture

After a certain depth, adding additional layers to feed-forward convNets results in a higher training error and higher validation error. When adding layers, performance increases only up to a certain depth, and then it rapidly decreases. In the **ResNet** (**Residual Network**) paper, the authors argued that this underfitting is unlikely due to the vanishing gradient problem, because this happens even when using the batch normalization technique. Therefore, they have added a new concept called **residual block**. The ResNet team added connections that can skip layers:

ResNet uses standard convNet and adds connections that skip a few convolution layers at a time. Each bypass gives a residual block.



Residual block

In the 2015 ImageNet ILSVRC competition, the winner was ResNet from Microsoft, with an error rate of 3.57%. ResNet is a kind of VGG in the sense that the same structure is repeated again and again to make the network deeper. Unlike VGGNet, it has different depth variations, such as 34, 50, 101, and 152 layers. It has a whopping 152 layers compared to AlexNet 8, VGGNet's 19 layers, and GoogLeNet's 22 layers. The ResNet architecture is a stack of residual blocks. The main idea is to skip layers by adding connections to the neural network. Every residual block has 3 x 3 convolution layers. After the last conv layer, a GAP layer is added. There is only one fully connected layer to classify 1,000 classes. It has different depth varieties, such as 34, 50, 101, or 152 layers for the ImageNet dataset. For a deeper network, say more than 50 layers, it uses the **bottleneck** features concept to improve efficiency. No dropout is used in this network.

Other network architectures to be aware of include:

- Network in Network
- Beyond ResNet
- FractalNet, an ultra-deep neural network without residuals

Summary

In this chapter, we learned about the different CNN architectures. These models are pre-trained existing models and differ in network architecture. Each of these networks is designed to solve a problem specific to its architecture. So, here we described their architectural differences.

We also understood how our own CNN architecture, as defined in the previous chapter, differs from these advanced ones.

In the next chapter, we will learn how these pre-trained models can be used for transfer learning.