CHAPTER 3

Image Classification Using LeNet

The journey of a thousand miles begins with one step.

—Lao Tzu

And you have taken that extraordinary step of learning Deep Learning. Deep Learning is an evolving field. From the basic Neural Networks, we have now evolved to complex architectures solving a multitude of business problems. Deep Learning powered Image Processing and computer vision capabilities are allowing us to create better cancer detection solutions, reduce pollution levels, implement surveillance systems, and improve consumer experience. Sometimes, a business problem demands a customized approach. We might design our own network to suit the business problem at hand and based on the image quality available to us. The network design will also consider the available computation power to train and execute the networks. The researchers across organizations and universities spent a good amount of time in collecting and curating the datasets, cleaning and analyzing them, designing architectures, training and testing them, and iterating to improve the performance. It takes a good amount of time and a lot of patience to make a path-breaking solution.

In the first two chapters, we have discussed the fundamentals of a Neural Network and created a Deep Learning solution using Keras and Python. From this chapter onward, we'll start discussing complex Neural

© Vaibhav Verdhan 2021
V. Verdhan, *Computer Vision Using Deep Learning*,
https://doi.org/10.1007/978-1-4842-6616-8_3

Network architectures. We will first introduce the LeNet architecture. We will go through the network design, various layers, activation functions, and so on. Then, we'll develop models for image classification use cases.

In particular, we'll cover the following topics in this chapter:

1. LeNet architecture and its variants
2. Design of the LeNet architecture
3. MNIST digit classification
4. German traffic sign classification
5. Summary

Welcome to the third chapter and all the very best!

3.1 Technical requirements

The code and datasets for the chapter are uploaded at the GitHub link `https://github.com/Apress/computer-vision-using-deep-learning/tree/main/Chapter3` for this book. We will use the Jupyter Notebook. For this chapter, a CPU is good enough to execute the code, but if required you can use Google Colaboratory. You can refer to the reference of the book for Google Colab.

Let's proceed with the Deep Learning architectures in the next section.

3.2 Deep Learning architectures

When we discuss a Deep Learning network, there are a few components which immediately come to our mind – number of neurons, number of layers, activation functions used, loss, and so on. All these parameters play a paramount role in the design of the network and its performance. When we refer to the depth in a Neural Network, it is the number of hidden layers in the network. With the improvement in computing power, the networks got deeper and the demand for computing power also increased.

Info While you might think that increasing the number of layers in the network will result in an increase in accuracy, that's not always the case. This is exactly what resulted in a new network called ResNet.

Using these base components, we can design our own network. Researchers and scientists across the globe have spent a huge amount of time and efforts in coming up with different Neural Network architectures. The most popular architectures are *LeNet-5, AlexNet, VGGNet, GoogLeNet, ResNet, R-CNN (Region-based CNN), YOLO (You Only Look Once), SqueezeNet, SegNet, GAN (Generative Adversarial Network),* and so on. These networks use different numbers of hidden layers, neurons, activation functions, optimization methods, and so on. Some architectures are better suited than others based on the business problem at hand.

In this chapter, we'll discuss the LeNet architecture in detail and then develop some use cases. We are starting with LeNet as it is one of the easier frameworks to understand and one of the pioneers in Deep Learning architectures. We'll also check the impact on the performance of our model with changes in various hyperparameters.

3.3 LeNet architecture

As we discussed in the last section, LeNet is the first architecture we are discussing in the book. It is one of the simpler CNN architectures. It has garnered significance because before it was invented, character recognition was a cumbersome and time-consuming process. The LeNet architecture was introduced in 1998, and it gained popularity when it was used to classify handwritten digits on bank checks in the United States.

The LeNet architecture has a few forms – *LeNet-1*, *LeNet-4*, and *LeNet-5*, which is the most cited and celebrated one. They were developed by Yann LeCun over a period of time. In the interest of space, we are examining LeNet-5 in detail, and the rest of the architectures can be understood using the same methodology.

We are starting with the basic LeNet-1 architecture in the next section.

3.4 LeNet-1 architecture

The LeNet-1 architecture is simple to understand. Let's look at the dimensions of its layers.

First layer: 28x28 input image

Second layer: Four 24x24 Convolutional Layer (5x5 size)

Third layer: Average pooling layer (2×2 size)

Fourth layer: Eight 12×12 convolutional layer (5×5 size)

Fifth layer: Average pooling layers (2×2 size)

And finally, we have the output layer.

Info When LeNet was introduced, the researchers did not propose max pooling; instead, average pooling was used. You are advised to test the solution by using both average and max pooling.

The LeNet-1 architecture is shown in Figure 3-1.

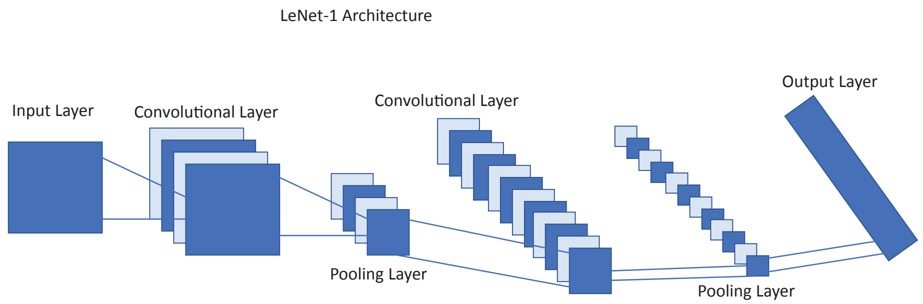


Figure 3-1. *LeNet-1 architecture is the first LeNet conceptualized. We should note how we have the first layer as a convolutional layer, followed by pooling, another convolutional layer, and a pooling layer. And finally, we have an output layer at the end*

You can see over here that we have the input layer, followed by the Convolutional Layer, then the Pooling Layer, followed by the Convolutional Layer, then the Pooling Layer, and then finally the output. The images are getting transformed during the entire network as per the configurations. We have explained in detail the function of all the layers and respective outputs while we discuss LeNet-5 in the subsequent sections.

3.5 LeNet-4 architecture

The LeNet-4 architecture is a slight improvement over LeNet-1. There is one more Fully Connected Layer and more feature maps.

First layer: 32x32 input image

Second layer: Four 24x24 Convolutional Layer (5x5 size)

Third layer: Average pooling layer (2×2 size)

Fourth layer: Sixteen 12×12 convolutional layer (5×5 size)

Fifth layer: Average pooling layers (2×2 size)

The output is fully connected to 120 neurons, which are fully connected further to 10 neurons as the final output.

The LeNet-4 architecture is shown in Figure 3-2.

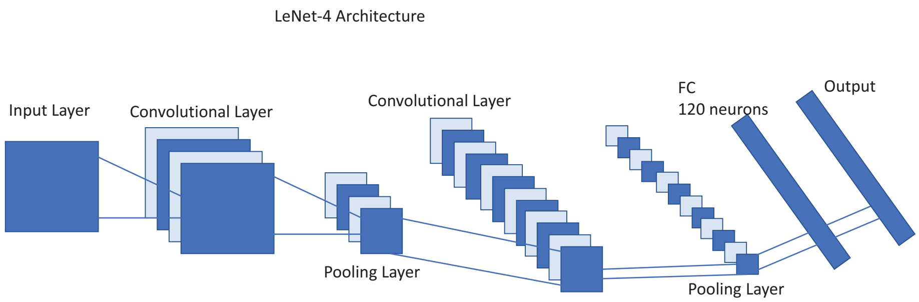


Figure 3-2. *LeNet-4 is an improvement over the LeNet-1 architecture. One more fully connected layer has been introduced in it*

For a detailed understanding of the function of all the layers, refer to the next section where we discuss LeNet-5.

3.6 LeNet-5 architecture

Between all three LeNet architectures, the most quoted architecture is LeNet-5, and it is the one which is generally used while solving the business problems.

The LeNet-5 architecture is shown in Figure 3-3. It was originally discussed in the paper "Gradient-Based Learning Applied to Document Recognition," Y. LeCun et al. The paper can be accessed at `http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf`.

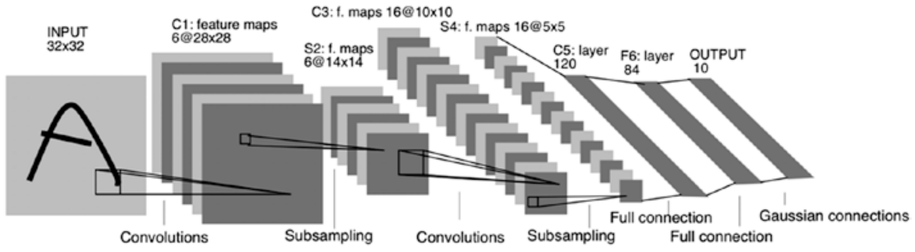


Figure 3-3. *LeNet architecture – the image has been taken from* *http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf*

Let's discuss its layers in detail as it's the most commonly used architecture:

1. First layer: The first layer of LeNet-5 is a 32x32 input image layer. It is a grayscale image that passes through a convolutional block with six filters of size 5x5. The resulting dimensions are 28x28x1 from 32x32x1. Here, 1 represents the channel; it's 1 because it's a grayscale image. If it was RGB, it would have been three channels.

2. Second layer: The pooling layer also called the *subsampling layer* has a filter size of 2x2 and a stride of 2. Image dimensions are reduced to 14x14x6.

3. Third layer: This is again a Convolutional Layer with 16 feature maps, a size of 5x5, and a stride of 1. Note that in this layer, only 10 out of 16 feature maps are connected to 6 feature maps of the previous layer. It gives us a few clear advantages:

a. The computation cost is lower. This is because the number of connections is reduced to 151,600 from 240,000.

b. The total number of training parameters for this layer is 1516 instead of 2400.

c. It also breaks the symmetry of the architecture, and hence the learning in the network is better.

4. Fourth layer: It is a pooling layer with a filter size of 2x2 and a stride of 2 with output as 5x5x16.

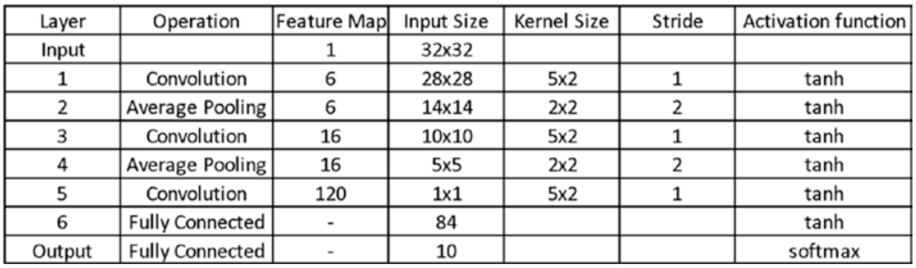
5. Fifth layer: It is a Fully Connected Convolutional layer with 120 feature maps and size being 1x1 each. Each of the 120 units is connected to 400 nodes in the previous layer.

6. Sixth layer: It is a Fully Connected layer with 84 units.

7. Finally, the output layer is a softmax layer with ten possible values corresponding to each digit.

A summary of the LeNet-5 architecture is shown in Table 3-1.

Table 3-1. *Summary of the entire LeNet architecture*



Layer	Operation	Feature Map	Input Size	Kernel Size	Stride	Activation function
Input		1	32x32			
1	Convolution	6	28x28	5x2	1	tanh
2	Average Pooling	6	14x14	2x2	2	tanh
3	Convolution	16	10x10	5x2	1	tanh
4	Average Pooling	16	5x5	2x2	2	tanh
5	Convolution	120	1x1	5x2	1	tanh
6	Fully Connected	-	84			tanh
Output	Fully Connected	-	10			softmax

LeNet is a small and very easy architecture to understand. Yet, it is large and mature enough to generate good results. It can be executed on a CPU too. At the same time, it will be a good idea to test the solution with different architectures and test the accuracy to choose the best one.

We have a boosted LeNet architecture which is being discussed next.

3.7 Boosted LeNet-4 architecture

Boosting is an ensemble technique that combines weak learners into strong ones by constantly improving from the previous iteration. In Boosted LeNet-4, the outputs of the architectures are added, and the one with the highest value is the predicted class.

The Boosted LeNet-4 architecture is shown in Figure 3-4.

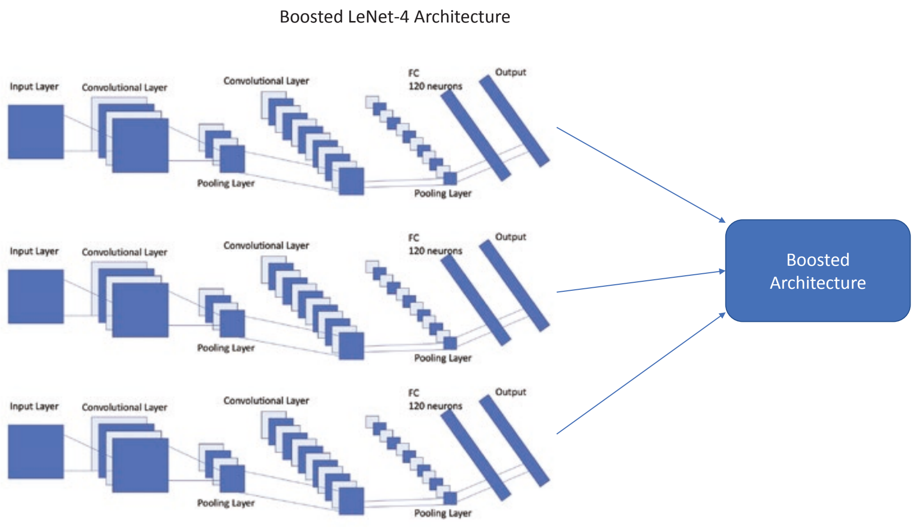


Figure 3-4. *Boosted LeNet-4 architecture combines weak learners to make an improvement. The final output is much more accurate and robust*

In the network architecture depicted earlier, we can see how the weak learners are combined to make a strong predictive solution.

LeNet is the first of the few architectures which became popular and were used to solve Deep Learning problems. With more progress and research, we have developed advanced algorithms, but still LeNet retains a special place.

It is time for us to create the first solution using the LeNet architecture.

3.8 Creating image classification models using LeNet

Now that we have understood the LeNet architecture, it is time to develop actual use cases with it. We are going to develop two use cases using the LeNet architecture. LeNet is a simple architecture so the code can be compiled on your CPU itself. We are making slight adjustments to the architecture in terms of the activation functions using the Max Pooling function instead of Average Pooling and so on.

Info Max pooling extracts the important features as compared to average pooling. Average pooling smooths the image, and hence sharp features might not be identified.

Before we start with the code, there is a small setting we should be aware of. The position of the channels in the tensor decides how we should reshape our data. This can be observed in step 8 of the following case study.

Each image can be represented by height, width, and the number of channels or number of channels, height, and width. If channels are at the first position of the input array, we reshape using the channels_first condition. It means that channels are in the first position in a tensor (n-dimensional array). And vice versa is true for channels_last.

3.9 MNIST classification using LeNet

This use case is a continuation of the MNIST dataset we used in the previous chapter. The code is checked in at GitHub link given at the start of the chapter

1. First, import all the required libraries.

```
import keras
from keras.optimizers import SGD
from sklearn.preprocessing import
LabelBinarizer from sklearn.model_selection
import train_test_split from sklearn.metrics
import classification_report from sklearn
import datasets
from keras import backend as K
import matplotlib.pyplot as plt
import numpy as np
```

2. Next, we import the dataset and then we import a series of layers from Keras.

```
from keras.datasets import mnist ## Data set is
imported here
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import
MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras import backend as K
```

3. Define the hyperparameters. This step is similar to the one in the previous chapter where we developed the MNIST and dog/cat classification.

```
image_rows, image_cols = 28, 28
batch_size = 256
num_classes = 10
epochs = 10
```

4. Load the dataset now. MNIST is the dataset that is added by default in the library.

```
(x_train, y_train), (x_test, y_test) =
mnist.load_data()
```

5. Convert the image data to float and then normalize it.

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32') x_train /= 255
x_test /= 255
```

6. Let's print the shape of our train and test datasets.

```
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

7. In the next block of code, we convert our variables into one-hot encoding. We use Keras's to_ categorical method for it.

```
y_train = keras.utils.to_categorical(y_train,
num_classes)
y_test = keras.utils.to_categorical(y_test,
num_classes)
```

Tip We are using print statements to analyze the output at each of the steps. It allows us to debug at a later stage if required.

8. Let's reshape our data accordingly.

```
if K.image_data_format() == 'channels_first':
   x_train = x_train.reshape(x_train.shape[0],
   1, image_rows, image_cols)
   x_test = x_test.reshape(x_test.shape[0], 1,
   image_rows, image_cols)
   input_shape = (1, image_rows, image_cols)
else:
   x_train = x_train.reshape(x_train.shape[0],
   image_rows, image_cols, 1)
   x_test = x_test.reshape(x_test.shape[0],
   image_rows, image_cols, 1)
   input_shape = (image_rows, image_cols, 1)
```

It's time to create our model!

9. Start with adding a sequential layer, followed by a Convolutional Layer and the Max Pooling layer.

```
model = Sequential()
model.add(Conv2D(20, (5, 5),
padding="same",input_shape=input_shape))
model.add(Activation("relu")) model.
add(MaxPooling2D(pool_size=(2, 2),
strides=(2, 2)))
```

10. We are now going to add multiple layers of the Convolutional layer, Max Pooling layers, and layers to flatten the data.

    ```
    model.add(Conv2D(50, (5, 5), padding="same"))
    model.add(Activation("relu")) model.
    add(MaxPooling2D(pool_size=(2, 2),
    strides=(2, 2)))
    model.add(Flatten()) model.add(Dense(500))
    model.add(Activation("relu"))
    ```

11. We add a Dense Layer followed by the softmax layer. Softmax is used for classification models. After that, we compile our model.

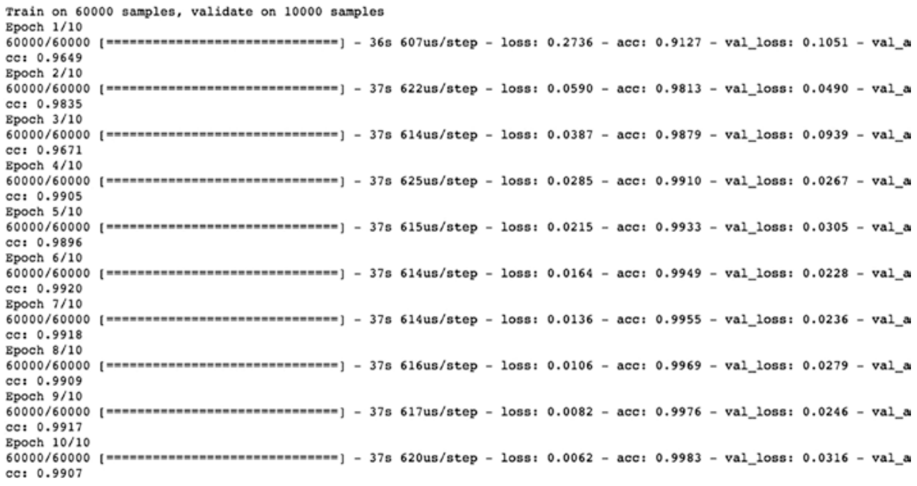
    ```
    model.add(Dense(num_classes)) model.
    add(Activation("softmax"))
    model.compile(loss=keras.losses.categorical_
    crossentropy, optimizer=keras.optimizers.
    Adadelta(), metrics=['accuracy'])
    ```

    The model is ready to be trained and fit.

12. We can see the impact per epoch on accuracy and loss. We encourage you to try with different variations of hyperparameters like epoch and batch size. Depending on the hyperparameters, the network will take time to process.

    ```
    theLeNetModel = model.fit(x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1, validation_data=(x_test, y_test))
    ```

Here, we can analyze how the loss and accuracy vary with each epoch.



```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 36s 607us/step - loss: 0.2736 - acc: 0.9127 - val_loss: 0.1051 - val_a
cc: 0.9649
Epoch 2/10
60000/60000 [==============================] - 37s 622us/step - loss: 0.0590 - acc: 0.9813 - val_loss: 0.0490 - val_a
cc: 0.9835
Epoch 3/10
60000/60000 [==============================] - 37s 614us/step - loss: 0.0387 - acc: 0.9879 - val_loss: 0.0939 - val_a
cc: 0.9671
Epoch 4/10
60000/60000 [==============================] - 37s 625us/step - loss: 0.0285 - acc: 0.9910 - val_loss: 0.0267 - val_a
cc: 0.9905
Epoch 5/10
60000/60000 [==============================] - 37s 615us/step - loss: 0.0215 - acc: 0.9933 - val_loss: 0.0305 - val_a
cc: 0.9896
Epoch 6/10
60000/60000 [==============================] - 37s 614us/step - loss: 0.0164 - acc: 0.9949 - val_loss: 0.0228 - val_a
cc: 0.9920
Epoch 7/10
60000/60000 [==============================] - 37s 614us/step - loss: 0.0136 - acc: 0.9955 - val_loss: 0.0236 - val_a
cc: 0.9918
Epoch 8/10
60000/60000 [==============================] - 37s 616us/step - loss: 0.0106 - acc: 0.9969 - val_loss: 0.0279 - val_a
cc: 0.9909
Epoch 9/10
60000/60000 [==============================] - 37s 617us/step - loss: 0.0082 - acc: 0.9976 - val_loss: 0.0246 - val_a
cc: 0.9917
Epoch 10/10
60000/60000 [==============================] - 37s 620us/step - loss: 0.0062 - acc: 0.9983 - val_loss: 0.0316 - val_a
cc: 0.9907
```

After ten epochs, the validation accuracy is 99.07%.

Now, let us visualize the results.

13. We will be plotting the training and testing accuracy in the next code block.

```
import matplotlib.pyplot as plt
f, ax = plt.subplots()
ax.plot([None] + theLeNetModel.history['acc'], 'o-')
ax.plot([None] + theLeNetModel.history['val_acc'], 'x-')
ax.legend(['Train acc', 'Validation acc'], loc = 0)
ax.set_title('Training/Validation acc per Epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('acc')
```

In the graph in Figure 3-5, we can see that with each subsequent epoch, the respective accuracy parameters for both training and validation continue to increase. After epoch 7/8, the accuracy stabilizes. We can test this with different values of hyperparameters.

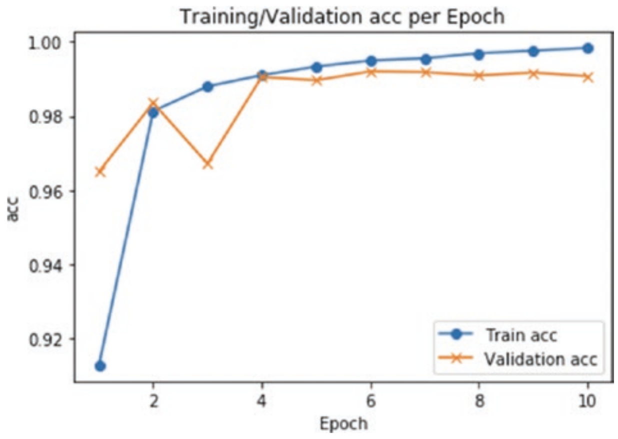


Figure 3-5. *Training and validation accuracy are shown here. After epoch 7/8, the accuracy has stabilized*

14. Let’s analyze the loss:

```
import matplotlib.pyplot as plt f,
ax = plt.subplots()
ax.plot([None] + theLeNetModel.history['loss'], 'o-')
ax.plot([None] + theLeNetModel.history['val_loss'], 'x-')
ax.legend(['Train loss', 'Validation loss'], loc = 0)
ax.set_title('Training/Validation loss per Epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('acc')
```

In the graph in Figure 3-6, we can see that with each subsequent epoch, the respective loss measures for both training and validation continue to decrease. After epoch 7/8, the accuracy stabilizes. We can test this with different values of hyperparameters.

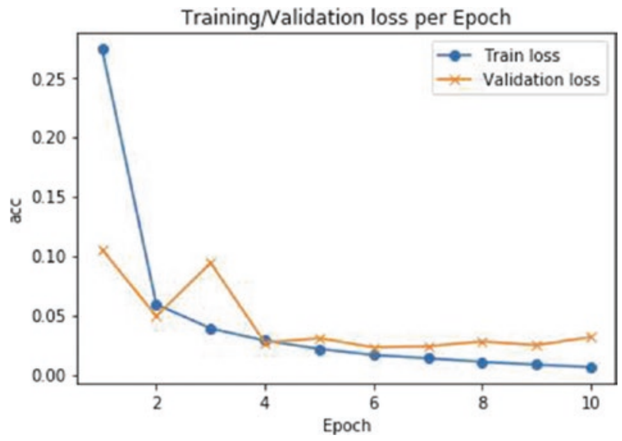


Figure 3-6. *Training and validation loss are shown here. After 7/8 epochs, the loss has stabilized and not much reductions are being observed*

Great! Now we have a working LeNet model to classify the images.

In this exercise, we trained an image classification model using the LeNet-5 architecture.

Info Use Google Colaboratory if you face any challenge with computation efficiency.

3.10 German traffic sign identification using LeNet

The second use case is the German traffic sign identification. It can be used in autonomous driving solutions.

In this use case, we are going to build a Deep Learning model using the LeNet-5 architecture.

1. We are going to follow a similar process throughout the book which is importing the libraries first.

```
import keras
from keras.optimizers import SGD
from sklearn.preprocessing import
LabelBinarizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn import datasets
from keras import backend as K
import matplotlib.pyplot as plt
import numpy as np
```

2. Import the Keras libraries along with all the packages required to create plots.

```
from keras.models import Sequential
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import
MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras import backend as K
```

3. Then import general libraries like numpy, matplotlib, os, OpenCV, and so on.

```
import glob
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import random
import matplotlib.image as mpimg
import cv2
import os
from sklearn.model_selection import train_test_
split
from sklearn.metrics import confusion_matrix
from sklearn.utils import shuffle
import warnings
from skimage import exposure
# Load pickled data
import pickle
%matplotlib inline matplotlib.style.
use('ggplot')
%config InlineBackend.figure_format = 'retina'
```

4. The dataset is provided as a pickle file and is saved as a train.p and test.p files. The dataset can be downloaded from Kaggle at www.kaggle.com/meowmeowmeowmeowmeow/gtsrb-german-traffic-sign.

```
training_file = "train.p"
testing_file = "test.p"
```

5. Open the files and save the data inside the train and test variables.

```
with open(training_file, mode='rb') as f: train
= pickle.load(f)
with open(testing_file, mode='rb') as f: test =
pickle.load(f)
```

6. Divide the dataset into test and train. We've taken a test size of 4000 here, but you are free to try different test sizes.

```
X, y = train['features'], train['labels']
x_train, x_valid, y_train, y_valid = train_
test_split(X, y, stratify=y,
test_size=4000, random_state=0)
x_test,y_test=test['features'],test['labels']
```

7. Let's have a look at some of the sample image files here. We have used the random function to select random images; thus, don't worry if your output is not the same as ours.

```
figure, axiss = plt.subplots(2,5, figsize=(15, 4))
figure.subplots_adjust(hspace = .2, wspace=.001)
axiss = axiss.ravel()
for i in range(10):
    index = random.randint(0, len(x_train))
    image = x_train[index]
    axiss[i].axis('off')
    axiss[i].imshow(image)
    axiss[i].set_title( y_train[index])
```

Here's the output as shown in Figure 3-7.

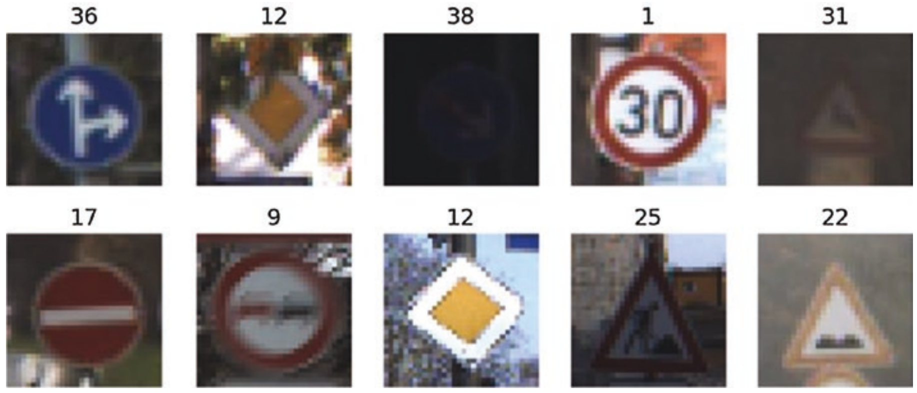


***Figure 3-7.** Some examples of the German traffic sign classification dataset are shown here*

8. Next, let's choose our hyperparameters. The number of distinct classes is 43. We have taken ten epochs to start with, but we encourage you to check the performance with different values of epochs. The same is true for batch size too.

```
image_rows, image_cols = 32, 32
batch_size = 256
num_classes = 43
epochs = 10
```

9. Now, we'll perform some exploratory data analysis. This is done to see how our image dataset looks and what are the frequency distributions of various classes in a histogram.

```
histogram, the_bins = np.histogram(y_train,
bins=num_classes) the_width = 0.7 * (the_
bins[1] - the_bins[0])
```

```
center = (the_bins[:-1] + the_bins[1:]) / 2
plt.bar(center, histogram, align='center',
width=the_width) plt.show()
```

The output is shown in Figure 3-8. We can observe that there is a difference in the number of examples of classes. A few classes are very well represented, while some are not. In a real-world solution, we would wish to have a balanced dataset. We are discussing more on it in Chapter 8 of the book.

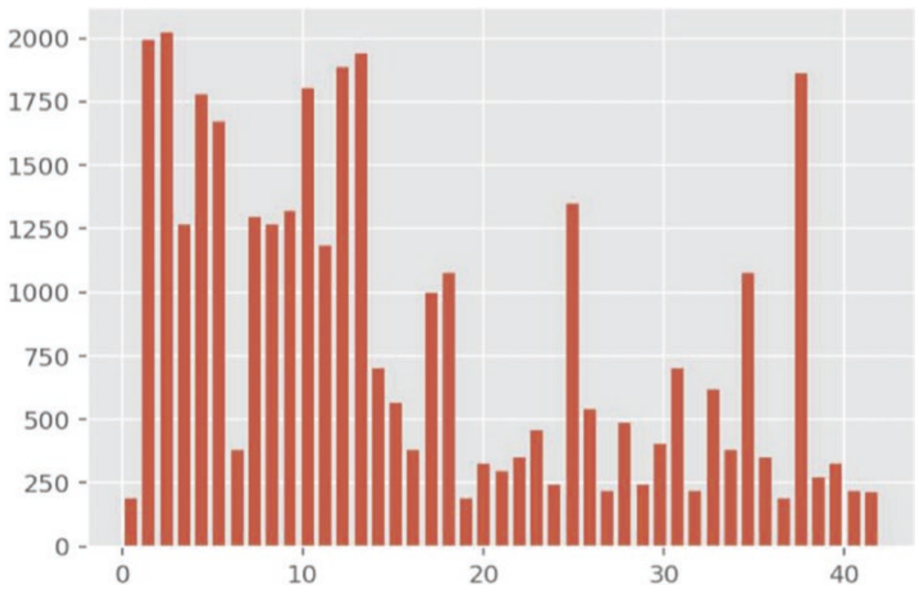


Figure 3-8. *Frequency distribution of various classes. Some classes have more examples, while some do not have much representation. Ideally, we should collect more data for less-represented classes*

10. Now, let us check how the distribution is across the different classes. It is a regular histogram function from the NumPy library.

```
train_hist, train_bins = np.histogram(y_train,
bins=num_classes)
test_hist, test_bins = np.histogram(y_test,
bins=num_classes)
train_width = 0.7 * (train_bins[1] - train_ bins[0])
train_center = (train_bins[:-1] + train_bins[1:]) / 2
test_width = 0.7 * (test_ bins[1] - test_bins[0])
test_center = (test_ bins[:-1] + test_bins[1:]) / 2
```

11. Now, plot the histograms; the color is set to red and green for train and test datasets, respectively.

```
plt.bar(train_center, train_hist,
align='center', color='red', width=train_width)
plt.bar(test_center, test_hist, align='center',
color='green', width=test_width)
plt.show()
```

Here's the output as shown in Figure 3-9.

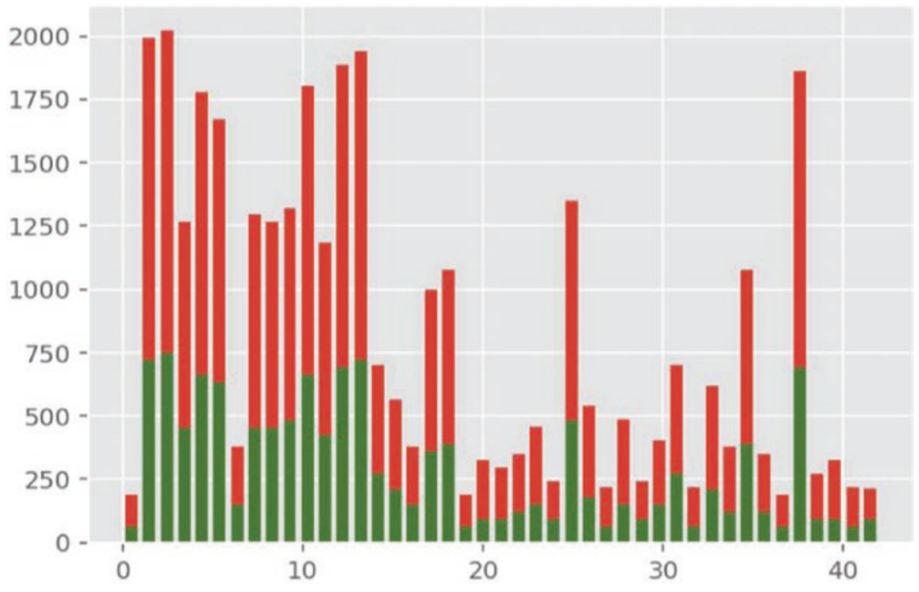


Figure 3-9. *Frequency distribution of various classes and distributed between train and test datasets. The train is shown in red color, while the test is depicted in green color*

Let's analyze the distribution here; look at the difference in the proportion of train vs. test in the preceding histogram.

12. Convert the image data to float and then normalize it.

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print(x_ train.shape[0], 'train samples')
print(x_test. shape[0], 'test samples')
```

So, we have 35,209 training data points and 12,630 testing ones. In the next step, convert class vectors to binary class matrices. This is similar to the steps in the last example where we developed the MNIST classification.

```
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

The following code block is the same as the one described in the MNIST classification developed earlier. Here, channels_first means that channels are at the first position in the array. And we are changing the input_shape depending on the position of channels_first.

```
if K.image_data_format() == 'channels_first':
    input_shape = (1, image_rows, image_cols)
else:
    input_shape = (image_rows, image_cols, 1)
```

Let us start creating the Neural Network Architecture now. The steps are similar to the previous use case.

13. Add a sequential layer followed by a Convolutional layer.

    ```
    model = Sequential() model.add(Conv2D(16,(3,3),
    input_shape=(32,32,3)))
    ```

14. Add the Pooling layer followed by Convolutional layers and so on.

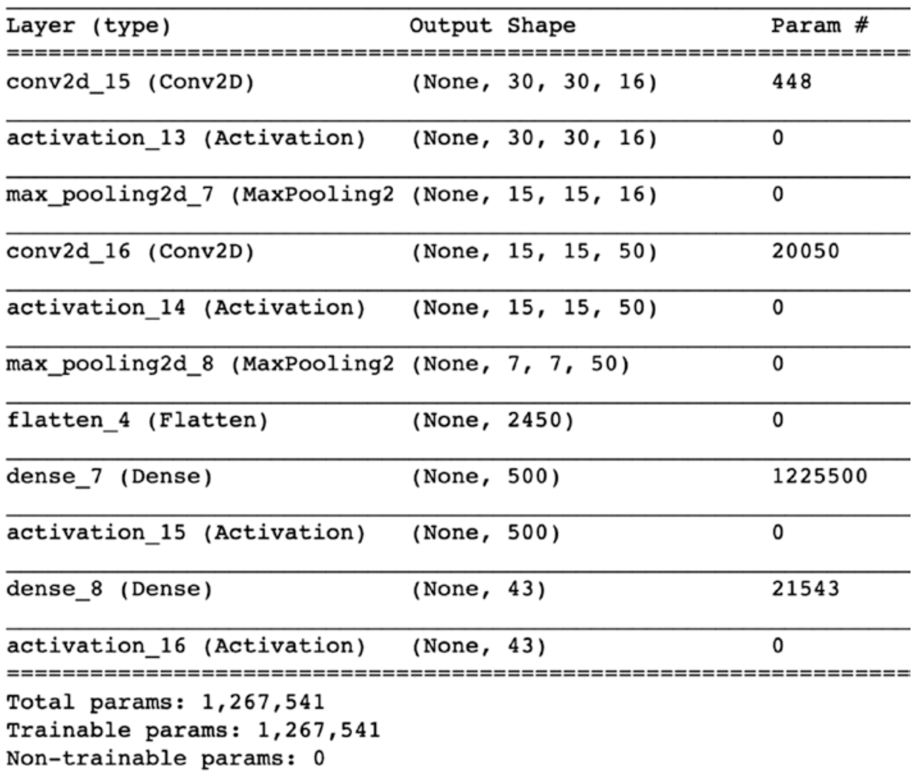
    ```
    model.add(Activation("relu")) model.
    add(MaxPooling2D(pool_size=(2, 2),
    strides=(2, 2)))
    ```

```
model.add(Conv2D(50, (5, 5), padding="same"))
model.add(Activation("relu")) model.
add(MaxPooling2D(pool_size=(2, 2),
strides=(2, 2)))
model.add(Flatten()) model.add(Dense(500))
model.add(Activation("relu"))
model.add(Dense(num_classes)) model.
add(Activation("softmax"))
```

15. Let us print the model summary.

```
model.summary()
```

Here's the output.

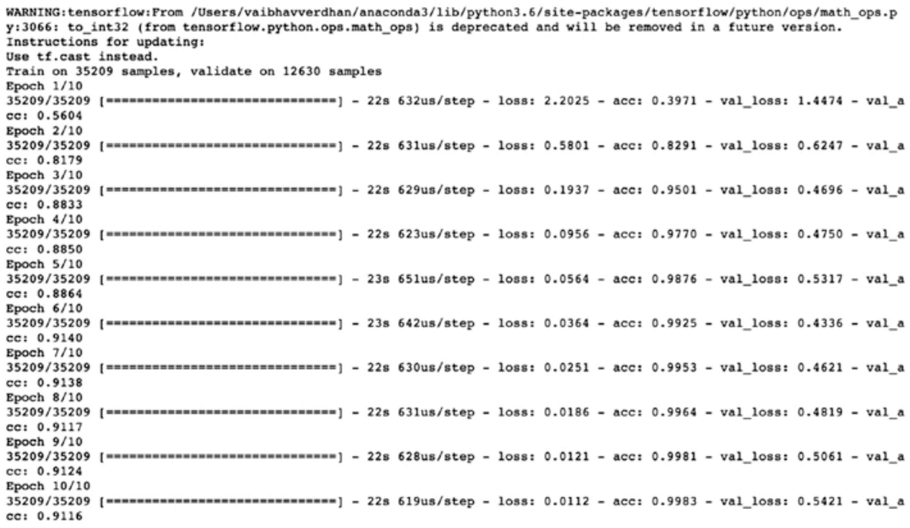


```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_15 (Conv2D)           (None, 30, 30, 16)        448
_________________________________________________________________
activation_13 (Activation)   (None, 30, 30, 16)        0
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 15, 15, 16)        0
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 15, 15, 50)        20050
_________________________________________________________________
activation_14 (Activation)   (None, 15, 15, 50)        0
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 7, 7, 50)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 2450)              0
_________________________________________________________________
dense_7 (Dense)              (None, 500)               1225500
_________________________________________________________________
activation_15 (Activation)   (None, 500)               0
_________________________________________________________________
dense_8 (Dense)              (None, 43)                21543
_________________________________________________________________
activation_16 (Activation)   (None, 43)                0
=================================================================
Total params: 1,267,541
Trainable params: 1,267,541
Non-trainable params: 0
```

16. The model is ready to be compiled; let's train it.

    ```
    model.compile(loss=keras.losses.categorical_
    crossentropy, optimizer=keras.optimizers.
    Adadelta(), metrics=['accuracy'])
    theLeNetModel = model.fit(x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
    verbose=1,
    validation_data=(x_test, y_test))
    ```

Here's the output as shown in Figure 3-10.



```
WARNING:tensorflow:From /Users/vaibhavverdhan/anaconda3/lib/python3.6/site-packages/tensorflow/python/ops/math_ops.p
y:3066: to_int32 (from tensorflow.python.ops.math_ops) is deprecated and will be removed in a future version.
Instructions for updating:
Use tf.cast instead.
Train on 35209 samples, validate on 12630 samples
Epoch 1/10
35209/35209 [==============================] - 22s 632us/step - loss: 2.2025 - acc: 0.3971 - val_loss: 1.4474 - val_a
cc: 0.5604
Epoch 2/10
35209/35209 [==============================] - 22s 631us/step - loss: 0.5801 - acc: 0.8291 - val_loss: 0.6247 - val_a
cc: 0.8179
Epoch 3/10
35209/35209 [==============================] - 22s 629us/step - loss: 0.1937 - acc: 0.9501 - val_loss: 0.4696 - val_a
cc: 0.8833
Epoch 4/10
35209/35209 [==============================] - 22s 623us/step - loss: 0.0956 - acc: 0.9770 - val_loss: 0.4750 - val_a
cc: 0.8850
Epoch 5/10
35209/35209 [==============================] - 23s 651us/step - loss: 0.0564 - acc: 0.9876 - val_loss: 0.5317 - val_a
cc: 0.8864
Epoch 6/10
35209/35209 [==============================] - 23s 642us/step - loss: 0.0364 - acc: 0.9925 - val_loss: 0.4336 - val_a
cc: 0.9140
Epoch 7/10
35209/35209 [==============================] - 22s 630us/step - loss: 0.0251 - acc: 0.9953 - val_loss: 0.4621 - val_a
cc: 0.9138
Epoch 8/10
35209/35209 [==============================] - 22s 631us/step - loss: 0.0186 - acc: 0.9964 - val_loss: 0.4819 - val_a
cc: 0.9117
Epoch 9/10
35209/35209 [==============================] - 22s 628us/step - loss: 0.0121 - acc: 0.9981 - val_loss: 0.5061 - val_a
cc: 0.9124
Epoch 10/10
35209/35209 [==============================] - 22s 619us/step - loss: 0.0112 - acc: 0.9983 - val_loss: 0.5421 - val_a
cc: 0.9116
```

Figure 3-10. *The accuracy and loss movement with respect to each epoch. We should note how the accuracy has improved from the first epoch to the last one*

After ten epochs, the validation accuracy is 91.16%.

Let us visualize the results now.

17. We will first plot the training and testing accuracy for the network.

```
import matplotlib.pyplot as plt
f, ax = plt.subplots()
ax.plot([None] + theLeNetModel.history['acc'], 'o-')
ax.plot([None] + theLeNetModel. history['val_acc'], 'x-')
ax.legend(['Train acc', 'Validation acc'], loc = 0)
ax.set_ title('Training/Validation acc per Epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('acc')
```

Here's the resulting plot as shown in Figure 3-11.

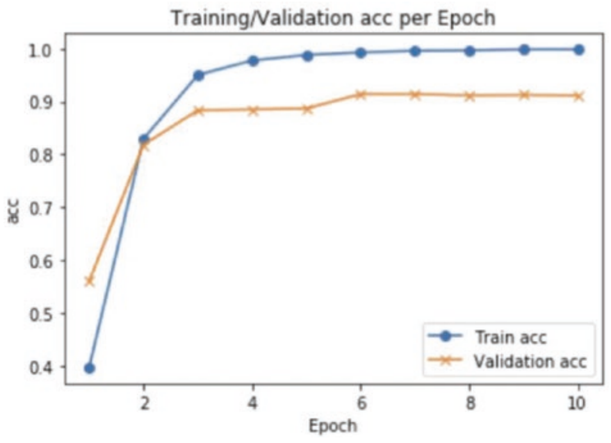


Figure 3-11. *Training and validation accuracy are shown here. After epoch 5/6, the accuracy has stabilized*

18. Let's plot the loss for training and testing data.

```
import matplotlib.pyplot as plt
f, ax = plt.subplots()
ax.plot([None] + theLeNetModel.history['loss'], 'o-')
ax.plot([None] + theLeNetModel. history['val_loss'], 'x-')
ax.legend(['Train loss', 'Validation loss'], loc = 0)
ax.set_ title('Training/Validation loss per Epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('acc')
```

The resulting plot can be seen in Figure 3-12.

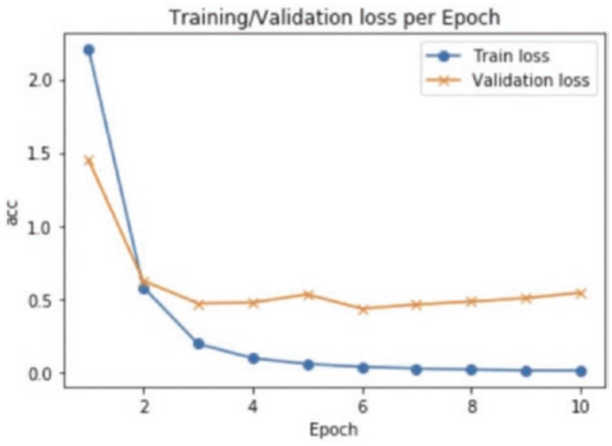


Figure 3-12. *Training and validation loss are shown here. After 5/6 epochs, the loss has stabilized and not much reductions are being observed*

The accuracy and loss function plots are generated to measure the performance of the model. The plots are similar to the ones developed in the MNIST classification model.

Info All the model's performance parameters are inside theLeNetModel.model or theLeNetModel.model.metrics.

In this example, we are taking one additional step and creating the Confusion Matrix for the prediction too. For this, we have to first make the predictions over the test set and then compare the predictions with the actual labels of the images.

19. Make the prediction using the predict function.

```
predictions = theLeNetModel.model.predict
(x_test)
```

20. Now, let us create the Confusion Matrix. It is available in the scikit-learn library.

```
from sklearn.metrics import confusion_matrix
import numpy as np
confusion = confusion_matrix(y_test,
np.argmax(predictions,axis=1))
```

21. Let us now create a variable called cm which is nothing but the confusion matrix.

    Feel free to print it and analyze the results.

```
cm = confusion_matrix(y_test,
np.argmax(predictions,axis=1))
```

22. Now let us start the visualization for the Confusion Matrix. Seaborn is another library along with matplotlib which is used for visualization.

```
   import seaborn as sn
df_cm = pd.DataFrame(cm, columns=np.unique
(y_test), index = np.unique(y_test))
```

```
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
plt.figure(figsize = (10,7))
sn.set(font_scale=1.4)#for label size
sn.heatmap(df_cm, cmap="Blues",
annot=True,annot_kws={"size": 16})# font size
```

The output is shown as follows. Due to the number of dimensions, the Confusion Matrix is not clearly visible in Figure 3-13, so let's make it a little better in the next code block.

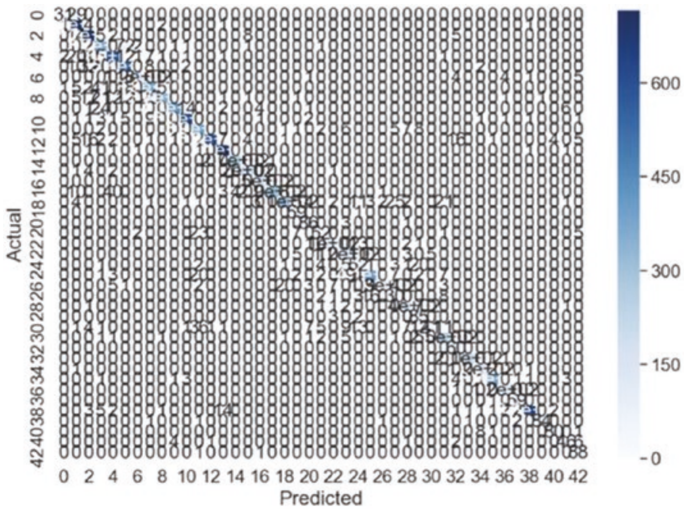


***Figure 3-13.** Confusion matrix is shown, but due to the number of dimensions, the output is not very clear which we are improving in the next figure*

23. Here, we are plotting the Confusion Matrix again. Please note that we have defined a function plot_confusion_matrix which takes a confusion matrix as the input parameter. Then we are using the regular matplotlib library and its functions to plot the Confusion Matrix. You can use this function for other solutions too.

    ```
    def plot_confusion_matrix(cm):
    cm = [row/sum(row) for row in cm]
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    cax = ax.matshow(cm, cmap=plt.cm.Oranges) fig.
    colorbar(cax)
    plt.title('Confusion Matrix') plt.
    xlabel('Predicted Class IDs') plt.ylabel('True
    Class IDs')
    plt.show()
       plot_confusion_matrix(cm)
    ```

Here's the plot that shows the confusion matrix (Figure 3-14).

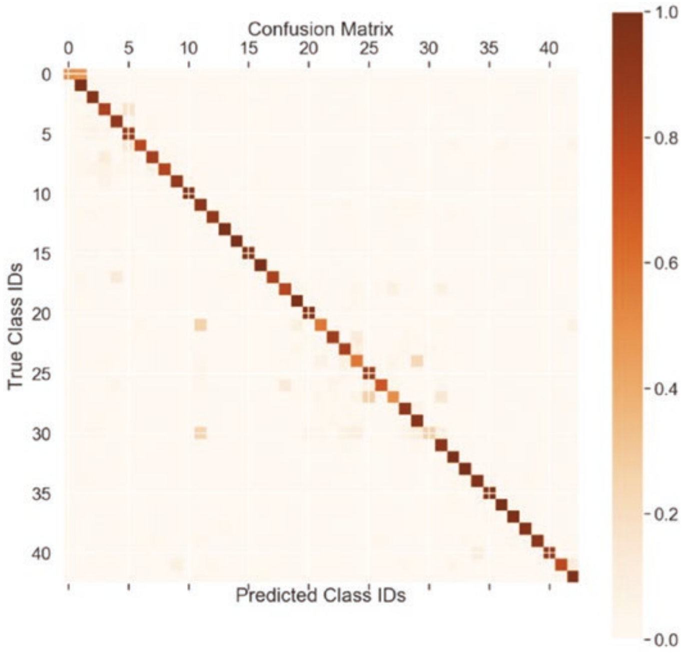


Figure 3-14. *Confusion matrix generated for all the classes. For a few classes, the results are not very good. It is advisable to scout for the misclassifications and analyze the reason*

We can see over here that for some classes we do have great results. You are advised to analyze the results and iterate with hyperparameters to see the impact. The observations which have misclassifications should be analyzed to find out the reason. For example, in the case of digit classification, an algorithm might become confused between 1 and 7. Hence, once we have tested the model, we should look for the misclassifications done and find out the reason. A potential solution is to remove the confusing images from the training dataset. Improving the quality of the images and increasing the quantity of the misclassified classes can also help in tackling the issue.

With this, we have completed the two case studies on image classification using LeNet. We are coming to the end of the chapter. You can proceed to the summary now.

3.11 Summary

Neural Network architectures are quite interesting and powerful solutions to computer vision problems. They allow us to train on a very large dataset and are useful to identify images correctly. This capability can be used for a very large variety of problems across domains. At the same time, it is imperative to note that the quality of the solution depends a lot on the quality of the training dataset. Remember the famous saying, garbage in garbage out.

We studied the concepts of convolutional, max pooling, padding, and so on in the last chapters and developed solutions using CNN. This chapter marks the start of customized Neural Network architectures. These architectures differ from each other by their design, that is, the number of layers, activation functions, strides, kernel size, and so on. More often, we test three of four distinct architectures to compare the accuracies.

In this chapter, we discussed LeNet architectures and focused on LeNet-5. We developed two use cases with end-to-end implementation right from data loading to designing of the network and testing the accuracy.

In the next chapter, we'll study another popular architecture called VGGNet.

You should be able to answer the questions in the exercise now!

REVIEW EXERCISES

1. What is the difference between different versions of LeNet?
2. How can we measure accuracy distribution with epoch?
3. We have discussed two use cases in the chapter. Iterate the same solution with different values of hyperparameters. Create a loss function and accuracy distribution for the use cases done in the last chapter.

4. Take the dataset used in the last chapter and test with LeNet-5 to compare the results.

5. Download the Image Scene classification dataset from www.kaggle.com/puneet6060/intel-image-classification/version/2. Execute the code used in the German traffic classification dataset for this dataset.

6. Download the Linnaeus 5 dataset from http://chaladze.com/l5/. It contains five classes – berry, bird, dog, flower, and other. Use this dataset to create a CNN-based solution.

3.11.1 Further readings

1. Go through the paper "Convolutional Neural Network for 3D object recognition using volumetric representation" at https://drive.google.com/drive/folders/1-5V1vj88-ANdRQJ5PpcAQkErvG7lqzxs.

2. Go through the paper for Alzheimer's disease classification using CNN at https://arxiv.org/abs/1603.08631.