Chapter 11

Neural Network

11.1 Introduction

Neural networks has taken the world by storm in the last decade. However, the work has been ongoing since the 1940s. Some of the initial work was in modeling the behavior of a biological neuron mathematically. Frank Rosenblat in 1958 built a machine that showed an ability to learn based on the mathematical notion of a neuron. The process of building neural networks were further refined over the next 4 decades. One of the most important papers that allowed training arbitrarily complex networks appeared in 1986 in a work by David E. Rumelhart, Geoffrey Hinton, and Ronald J. Williams [RHW86]. This paper re-introduced the back-propagation algorithm that is the workhorse of the neural network as used today. In the last 2 decades, due to the availability of cheaper storage and compute, large networks have been built that solve significant practical problems. This has made the neural network and its cousins such as the convolution neural network, recurrent neural network, etc., household names.

In this chapter, we will begin the discussion with the mathematics behind neural networks, which includes forward and back-propagation. We will then discuss the visualization of a neural network. Finally, we will discuss building a neural network using Keras, a Python module for machine learning and deep learning.

Interested readers are recommended to follow the discussions in the following sources: [Dom15], [MTH], [GBC16], [Gro17].

11.2 Introduction

A neural network is a non-linear function with many parameters. The simplest curve is a line with two parameters: slope and intercept. A neural network has many more parameters, typically in the order of 10,000 or more and sometimes millions. These parameters can be determined by the process of optimizing a loss function that defines the goodness of fit.

11.3 Mathematical Modeling

We will begin the discussion of the mathematics of a neural network by fitting lines and planes. We can then extend it to any arbitrary curve.

11.3.1 Forward Propagation

The equation of a line is defined as

$$y_1 = Wx + b \tag{11.1}$$

where x is the independent variable, y_1 is the dependent variable, W is the slope of the line and b is the intercept. In the world of machine learning, W is called the weight and b is the bias.

If the independent variable x is a scalar, then Equation 11.1 is a line, W is scalar and b is a scalar. However, if x is a vector, then Equation 11.1 is a plane, W is a matrix and b is a vector. If x is very large, Equation 11.1 is called a hyper-plane. Equation 11.1 is a linear equation and the best model that can ever be created using it would be a linear model as well.

In order to create non-linear models in a neural network, we add a non-linearity to this linear model. We will discuss one such non-linearity

called sigmoid. In practice however, other non-linearities such as tanh, rectified linear unit (RELU), and leaky RELU are also used.

The equation of a sigmoid function is

$$y = \frac{1}{1 + e^{-x}} \tag{11.2}$$

When x is a large number, the value of y asymptotically reaches 1 (11.1), while for small values of x, the value of y asymptotically reaches 0. In the region along the x-axis between -1 and $+1$ approximately, the curve is linear and is non-linear everywhere else.

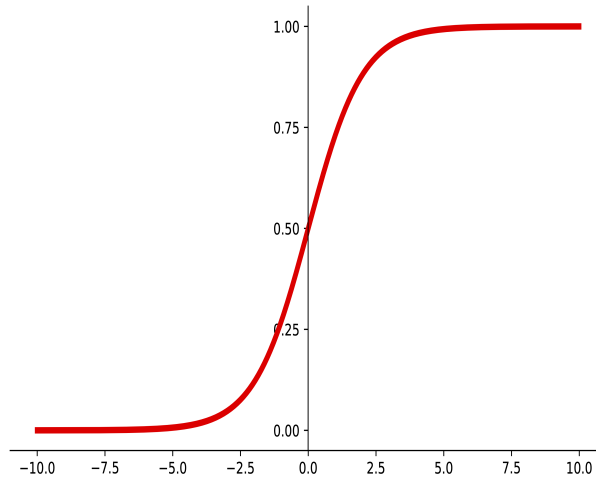


FIGURE 11.1: Sigmoid function.

If the y_1 from 11.1 is passed through a sigmoid function, we will obtain a new y_1,

$$y_1 = \frac{1}{1 + e^{-W_1 x - b_1}} \tag{11.3}$$

where W_1 is the weight of the first layer and b_1 is the bias of the first layer. The new y_1 in Equation 11.3 is a non-linear curve. The equation can be rewritten as,

$$y_1 = \sigma(W_1 x + b_1) \tag{11.4}$$

In a simple neural network here, we will add another layer (i.e., another set of W and b) to which we will pass the y_1 obtained from Equation 11.4.

$$y = W_2 y_1 + b_2 \tag{11.5}$$

where W_2 is the weight of the second layer and b_2 is the bias of the second layer.

If we substitute 11.4 in 11.5, we obtain,

$$y = W_2 \sigma(W_1 x + b_1) + b_2 \tag{11.6}$$

We can repeat this process by adding more layers and create a complex non-linear curve. However, for clarity sake, we will limit ourselves to 2 layers.

In Equation 11.6, there are 4 parameters namely, W_1, b_1, W_2, and b_2. If x is a vector, then W_1 and W_2 are matrices and b_1 and b_2 are vectors. The aim of a neural network is to determine the value inside these matrices and vectors.

11.3.2 Back-Propagation

The value of the 4 parameters can be determined using the process of back-propagation. In this process, we begin by assuming an initial value for the parameters. They can be assigned a value of 0 or some random value could be used.

We will then determine the initial value of y using Equation 11.6. We will denote this value as $\hat{y}$. The actual value y and the predicted

value $\hat{y}$ will not be equal. Hence there will be an error between them. We will call this error "loss."

$$L = (\hat{y} - y)^2 \tag{11.7}$$

Our aim is to minimize this loss by finding the correct value for the parameters of Equation 11.6.

Using the current value of the parameter, its new value iteratively can be calculated using,

$$W_{new} = W_{old} - \epsilon \frac{\partial L}{\partial W} \tag{11.8}$$

where W is a parameter and L is the loss function. This equation is generally called an 'update equation'.

To simplify the calculation of partial derivatives such as $\frac{\partial L}{\partial W}$, we will derive them in parts and assemble them using chain rule.

We will begin by calculating $\frac{\partial L}{\partial \hat{y}}$ using Equation 11.7

$$\frac{\partial L}{\partial \hat{y}} = 2 * (\hat{y} - y) \tag{11.9}$$

Then we will calculate $\frac{\partial L}{\partial W_2}$ using the chain rule,

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial W_2} \tag{11.10}$$

If we substitute $\hat{y}$ from Equation 11.5 and $\frac{\partial L}{\partial \hat{y}}$ from Equation 11.9, we obtain,

$$\frac{\partial L}{\partial W_2} = 2 * (\hat{y} - y) * y_1 \tag{11.11}$$

The partial derivative can then be used to update the value of W_2 using the existing value of W_2 with the help of the update equation. A similar calculation (left as an exercise to the reader) can be shown for b_2 as well.

Next we will calculate the new value of W_1,

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial y_1} \frac{\partial y_1}{\partial W_1} \tag{11.12}$$

which can be computed using Equations 11.9, 11.5 and 11.4 respectively. Thus,

$$\frac{\partial L}{\partial W_1} = 2(\hat{y} - y)(W_2)(\sigma(W_1 x + b_1)(1 - \sigma(W_1 x + b_1))x) \tag{11.13}$$

which can be simplified to

$$\frac{\partial L}{\partial W_1} = 2xW_2(\hat{y} - y)\sigma(W_1 x + b_1)(1 - \sigma(W_1 x + b_1)) \tag{11.14}$$

The new value of W_1 can be calculated using the update Equation 11.8. A similar calculation (left as an exercise to the reader) can be shown for b_1 as well.

For every input data point or a batch of data points, we perform forward propagation, determine the loss, and then back-propagate to update the parameters (weights and biases) using the update equation. This process is repeated with all the available data.

In summary, the process of back-propagation finds the partial derivatives of the parameters of a neural network system and uses the update equation to find a better value for the parameters by minimizing the loss.

11.4 Graphical Representation

Typically, a neural network is represented as shown in Figure 11.2. The left layer is called the input layer, the middle is called the hidden layer, and the right is called the output layer.

A node (filled circle) in a given layer is connected to all the nodes in the next layer but is not connected to any nodes in that layer. In Figure 11.2, arrows are only drawn to originate from an input layer to the first node in the hidden layer. For clarity sake, the lines ending on other nodes are omitted. The values at each of the input nodes is multiplied with the weights in the line between the nodes. The weighted inputs are then added in the node in the hidden layer and passed through the sigmoid function or any other non-linearity. The output of the sigmoid function is then weighted in the next layer and the sum of all those weights will be the output of the output layer ($\hat{y}$).

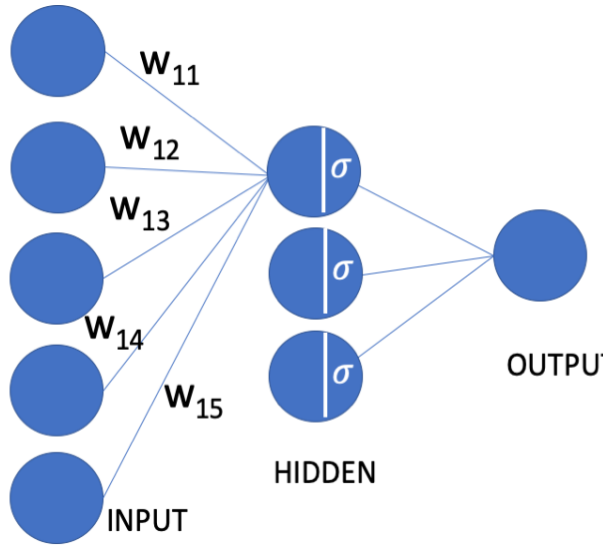


FIGURE 11.2: Graphical representation of a neural network.

If there are n nodes in the input layer and m nodes in the hidden layer, then the number of edges connecting from the input to hidden-layer will be n*m. This can be represented as a matrix of size [n, m]. Then the operation described in the previous paragraph will be a dot product between the input x and the matrix followed by application of the sigmoid function described in Equation 11.4. This matrix is the W_1 we have previously described.

If there are m nodes in the hidden layer and k nodes in the output layer, then the number of edges connecting from hidden to output will be m*k. This can be represented as a matrix of size [m, k] and is the matrix W_2 we have previously described.

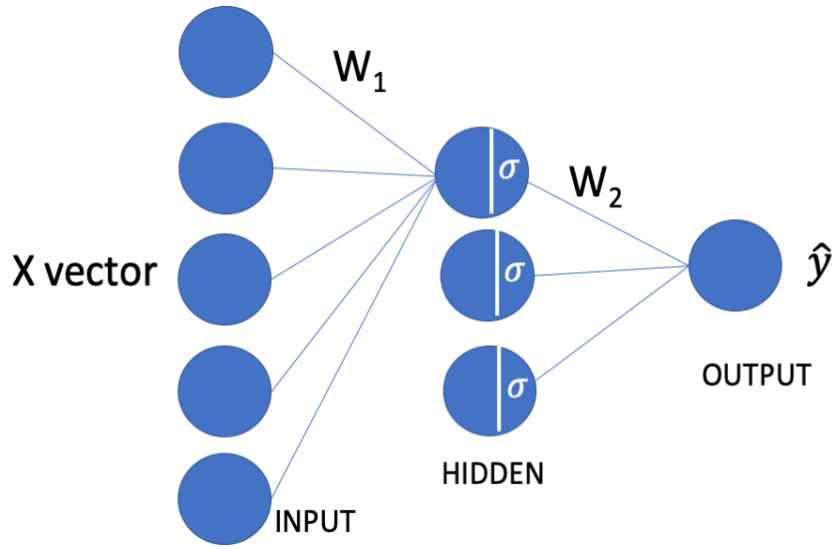


FIGURE 11.3: Graphical representation of a neural network as weight matrices.

During the forward propagation, a value of x is used as an input to begin the compute that is propagated from the input side to the output. The loss is calculated by comparing the predicted value and the actual. The gradients are then computed in reverse from the output layer toward the input and the parameters (weights and biases) are updated by back-propagation.

In this discussion, we assumed that y is a continuous function and its value is a real number. This class of problem is called a regression problem. An example of such a problem is the prediction of price of an item based on images.

11.5 Neural Network for Classification Problems

The other class of problem is the classification problem where the dependent variable y takes discrete values. An example of such a problem is identifying a specific type of lung cancer given an image. There are two major types: small cell lung cancer (SCLC) and non-small cell lung cancer (NSCLC).

In a classification problem, we aim to draw a boundary between two classes of points as shown in Figure 11.4. The two classes of points in the image are the circles and the plusses. A linear boundary (such as a line or plane) that has the lowest error cannot be drawn between these two sets of points. A neural network can be used to draw a non-linear boundary.

One of the common loss functions for the classification problem is the cross entropy loss. It is defined as

$$L = -\sum ylog\hat{y} \tag{11.15}$$

where y is the actual value and $\hat{y}$ is the predicted value.

Since the loss function is different compared to the regression problem, the derivatives such as $\frac{\partial L}{\partial W}$ would yield a different equation compared to the one derived for the regression problem. However the approach remains the same.

11.6 Neural Network Example Code

The current crop of popular deep learning packages such as Tensorflow [ABC+16], Keras [C+20], etc., require the programmer to define the forward propagation while the back-propagation is handled by the package.

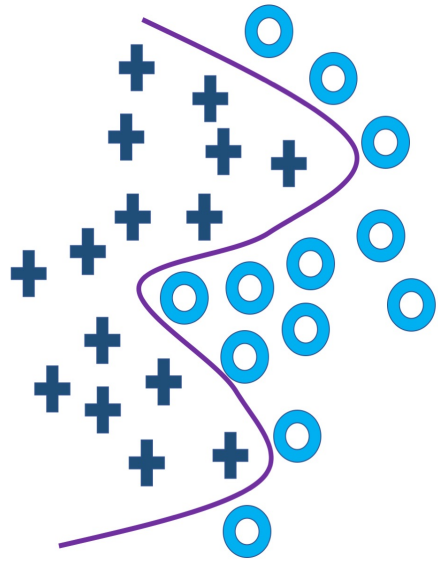


FIGURE 11.4: The neural network for the classification problem draws a non-linear boundary between two classes of points.

In the example below, we define a neural network to solve the problem of identifying handwritten digits from MNIST dataset [LCB10], a popular image dataset for benchmarking machine learning and deep learning applications. Figure 11.5 shows a few representative images from the MNIST dataset. Each image is 28 pixels by 28 pixels in size. The total number of pixels = 28*28 = 784. The images contain a single hand-drawn digit. As can be seen in the image, two numbers may not look the same in two different images. The task is to identify the digit in the image, given the image itself. The image is the input and the output is one of the 10 classes (number between 0 and 9).

We begin by importing all the necessary modules in Keras, specifically the Sequential model and Dense layer. The Sequential model allows defining a set of layers. In the mathematical discussion, we defined 2 layers. In Keras, these layers can be defined using the Dense layers class. A stack of these layers constitutes a Sequential layer.

We load the MNIST dataset using the convenient functionality (keras.datasets.mnist.load_data) available in Keras. This loads both the

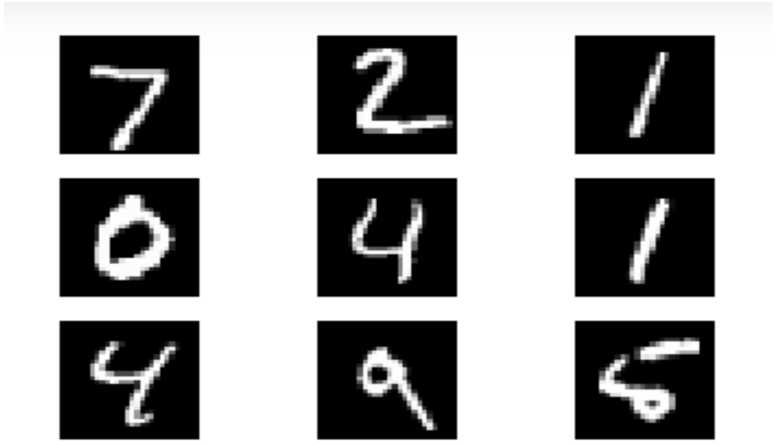


FIGURE 11.5: Some sample data from the MNIST dataset [LCB10].

training data as well as testing data. The number of images in the training dataset is 60,000 and the number of images in the testing dataset is 10,000. Each image is stored as a 784-pixel-long vector with 8-bit precision (i.e., pixel values are between 0 and 255). The corresponding y for each image is a single number corresponding to the digit in that image.

We then normalize the image by dividing each pixel value by 255 and subtracting 0.5. Hence the normalized image will have pixel values between -0.5 and $+0.5$.

The model is built by passing 3 Dense layers to the Sequential class. The first layer has 64 nodes, the second layer has 64 nodes. The first 2 layers use the Rectified Linear Unit (RELU) activation function for non-linearity. The last layer produces a vector of length 10. This vector is passed through a softmax function (Equation 11.16). The output of a softmax function is a probability distribution as each of the values corresponds to the probability of a given digit and also the sum of all the values in the vector equates to 1. Once we obtain this vector, determining the corresponding digit can be accomplished by finding the position in the vector with the highest probability value.

$$s_i = \frac{e^{x_i}}{\sum_i e^{x_i}} \tag{11.16}$$

We will pass the model through an optimization process by calling the fit function. We run the model through 5 epochs, where each epoch is defined as visiting all images in the training dataset. Typically we feed a batch of images for training instead of one image at a time. In the example, we use a batch of 32, which implies in each training a random batch of 32 images and the corresponding labels are passed.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from keras.datasets import mnist

# Fetch the train and test data.
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the image so that all pixel values
# are between -0.5 and +0.5.
x_train = (x_train / 255) - 0.5
x_test = (x_test / 255) - 0.5

# Reshape the train and test images to size 784 long vector.
x_train = x_train.reshape((-1, 784))
x_test = x_test.reshape((-1, 784))

# Define the neural network model with 2 hidden layer
# of size 64 nodes each.
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax'),
])
```

```
# Compile the model using Adam optimizer and use
# the cross entropy loss.
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model.
model.fit(x_train, to_categorical(y_train), epochs=5,
          batch_size=32)
```

The output contains the result of training 5 epochs. As can be seen the value of cross entropy loss decreases as the training progresses. It started at 0.3501 and finally ended at 0.0975. Similarly, the accuracy increased as the training progressed from 0.8946 to 0.9697.

```
Epoch 1/5
60000/60000 [===] - 3s 58us/step - loss: 0.3501 - accuracy:
0.8946
Epoch 2/5
60000/60000 [===] - 3s 56us/step - loss: 0.1790 - accuracy:
0.9457
Epoch 3/5
60000/60000 [===] - 3s 55us/step - loss: 0.1357 - accuracy:
0.9576
Epoch 4/5
60000/60000 [===] - 3s 55us/step - loss: 0.1129 - accuracy:
0.9649
Epoch 5/5
60000/60000 [===] - 3s 57us/step - loss: 0.0975 - accuracy:
0.9697
```

Interested readers must consult the Keras documentation for more details.

11.7 Summary

- Neural networks are universal function approximators. In training a neural network, we fit a non-linear curve using available data.
- To obtain non-linearity in a neural network, we combine a linear function with non-linear functions such as sigmoid, RELU, etc.
- The parameters of the non-linear curve are learnt through the process of back-propagation.
- Neural networks can be used for both regression and classification problems.

11.8 Exercises

1. You are given a neuron that performs addition of y = x1*w1+x2*w2, where x1 and x2 are the inputs and w1 and w2 are weights. Write the back-propagation equation for it. Also write the update equation for w1 and w2.
2. In a neural network, we combine linear function $Wx + b$ with a non-linear function. We stack these layers together to produce an arbitrarily complex non-linear function. What would happen if we do not use a non-linear function but still stack layers? What kind of curve can we build?
3. Why is sigmoid no longer popular as an activation function? Conduct research on this topic.

Chapter 12

Convolutional Neural Network

12.1 Introduction

The convolution neural network (CNN) is a biologically inspired mathematical model of vision. The journey began successfully with the work by David Hubel and Torsten Wiesel who won the 1981 Nobel prize in Physiology or Medicine for this work. The work by Hubel and Weisel was best summarized by the Nobel committee's press release ([ppr20]) from 1981. The following paragraph is a reproduction from the press release:

"... the visual cortex's analysis of the coded message from the retina proceeds as if certain cells read the simple letters in the message and compile them into syllables that are subsequently read by other cells, which, in turn, compile the syllables into words, and these are finally read by other cells that compile words into sentences that are sent to the higher centers in the brain, where the visual impression originates and the memory of the image is stored."

As the quote indicates, Hubel and Wiesel found that the brain has a series of neurons. The neurons nearest to the retina detect simple shapes such as lines in different orientation. The neurons next to detect complex shapes like curves. The neurons downstream detect more complex shapes like nose, ear, etc.

The understanding of the brain's visual cortex paved the way for mathematical modeling of the visual pathway. The first successful work was done by Kunihiko Fukushima ([Fuk80]). He demonstrated a

hierarchical model using convolution and downsampling. The convolution allowed viewing of only a part of the image or video while processing. The downsampling was performed by averaging. Many years later, a different method called "maxpooling" was introduced which is still in use today and will be discussed later in this chapter. The next major breakthrough was the work of Yann Lecun [LBD+89] who introduced a back-propagation approach to learn the parameters of a CNN.

With the availability of large quantities of data, cheaper storage, compute power and software, CNNs have become a go-to tool for solving image processing and computer vision problems in all areas of science and engineering.

12.2 Convolution

We discussed the process of applying convolution to an image in Chapter 4. In this section, we will discuss convolution from the perspective of a CNN. In the example in Chapter 4, the convolution was performed using a 5-by-5 filter where each element in the filter has a value of $\frac{1}{25}$. In a CNN, the values in the filter are determined by the learning process (i.e., the back-propagation process).

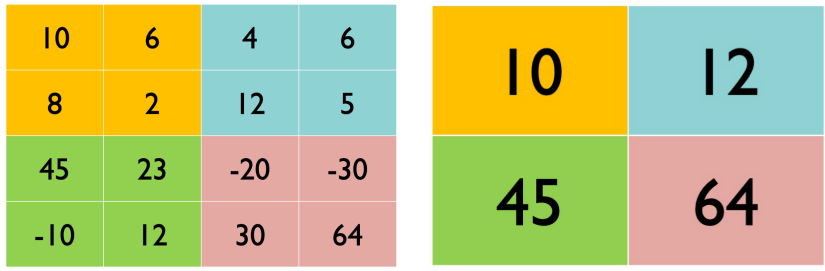
Also, unlike the examples from Chapter 4, in a CNN there are multiple filters used. These filters are arranged into layers. The first layer is designed to detect simple objects such as lines. Since there are many possible configurations (slopes) for lines, the first layer may have multiple filters to detect lines at all these orientations. The second layer is designed to detect curves. Since there are more configurations for a curve compared to a line, the number of filters in the second layer is typically more than the number of filters in the first layer. Modern CNNs[1] typically have more than 2 layers.

[1] CNNs are only 40 years old. We are distinguishing CNN architectures from the last 10 years from the ones by using the word modern.

12.3 Maxpooling

Maxpooling is a dimensionality reduction technique. It takes an input such as an image and reduces its size using the maximum among its neighbors. The effect of replacing a pixel value with its neighborhood maximum produces an abstracted representation of the image. We will demonstrate with an example.

Let's consider a small image (Figure 12.1(a)) of size 4x4 and also consider a maxpooling filter of size 2x2 placed on the top left corner of the image. In the maxpooling process, we will find the maximum value of the 2x2 region containing the values 10, 6, 8 and 2. We will create a new image (Figure 12.1(b)) where we will use the maximum value (10) from the previous calculation. We will then move this maxpooling filter by 2 steps (called a stride) over the image and find the next region on the image consisting of values 4, 6, 12 and 5. Its maximum value of 12 will be used for the next pixel in the output image. Once a row of maxpooling operation is completed, we move 2 rows below and continue this process.



(a) Input image. (b) Maxpooled image.

FIGURE 12.1: An example of applying maxpooling on a sub-image.

If an image is of size NxN and we move with a stride of 2x2, then the output image will be of size $\frac{N}{2}$x$\frac{N}{2}$. A higher stride can be used to reduce the image further.

The output from the convolution and maxpooling layer is finally passed to a classifier or a regressor, which is typically built using a neural network as discussed in the previous chapter. This is due to the fact that the convolution and maxpooling layer are data conditioners that prepare the data for a final classifier or a regressor.

12.4 LeNet Architecture

We will use the convolution layer and maxpooling to build LeNet [LBBH98], one of the first CNNs to revolutionize the field of computer vision. An input image (Figure 12.2) is passed to a series of 6 convolutions in the first layer. The output of the first convolution layer is subsampled using maxpooling, and is then passed to the second convolution layer, which contains 16 filters. The output of the second convolution layer is passed to the second maxpooling layer. The output of this layer is then flattened to a vector and passed to a neural network-based classifier or regressor.

Ideally, any classifier such as a Support Vector Machine SVM or Logistic regression can be used. However, a neural network, as discussed in the last chapter is preferred.

In the last chapter, we discussed that the parameters of the system can be learned using the process of back-propagation. The same applies to parameters (the values in the filter) of a CNN as well.

For every input image or a batch of images, we perform forward propagation through the convolution, maxpooling, and the neural network layers. We then determine the loss. Then we back-propagate through the neural network layers followed by back-propagation through the convolution layers and update the parameters (weights

and biases) using the update equation. This process is repeated with all the available data.

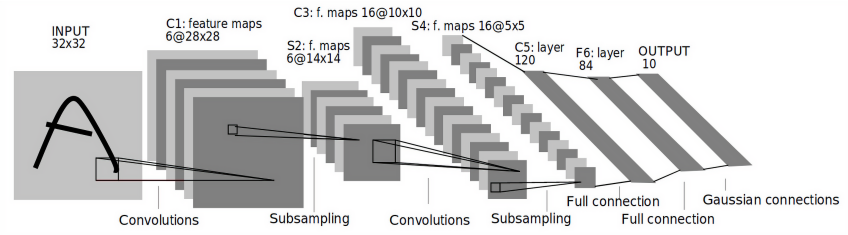


FIGURE 12.2: LeNet architecture diagram.

In the example below, we define a LeNet CNN to solve the problem of identifying handwritten digits from MNIST data set, which we also used in the last chapter.

We begin by importing all the necessary functionalities in Keras, specifically the Sequential model, Dense layer, Conv2D layer, and MaxPooling2D layer. The Sequential model allows defining a set of layers. The list of layers in order is shown in Figure 12.3.

The first layer is the input layer that takes a 28x28x1 image. The second layer is the convolution layer with 32 filters. The third layer is the maxpooling layer that reduces the image size by $\frac{1}{2}$. The fourth and fifth layer are the second set of a convolution layer with 64 filters and a second maxpooling layer that reduces the image size by $\frac{1}{2}$. The second maxpooling layer output image is flattened and passed through two neural network layers to produce an output prediction.

In this example, we use images of size 28x28x1. After passing through the first convolution layer, we will obtain a 3D volume of size 28x28x32. The first maxpooling layer reduces the size by $\frac{1}{2}$ to 14x14x32. This volume is passed through the second convolution layer which produces a volume of size 14x14x64. The second maxpooling layer reduces the image to 7x7x64. The flattened vector will thus be of size 3136 which is the product of 7, 7 and 64.

In the code, we load the training and test dataset by using the 'mnist.load_data() method'. The x values (image pixels) are normalized

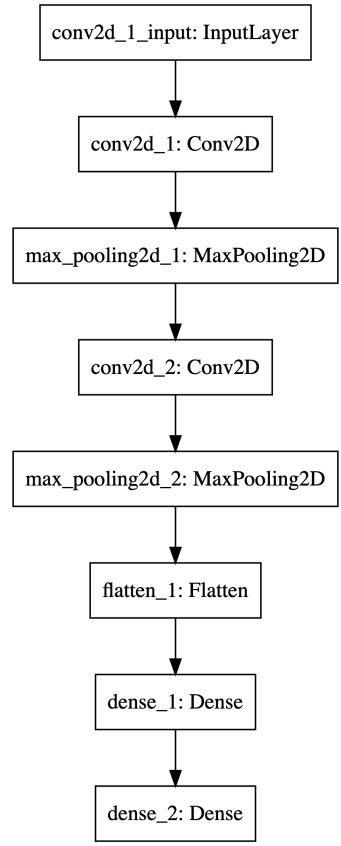


FIGURE 12.3: LeNet Keras model.

to be within the range $[-0.5, 0.5]$. They are then reshaped from its original shape of 50000x784 to 50000x28x28x1.

A sequential model is created and the various layers are added as described earlier. In all the layers, RELU non-linearity is added. The last layer is passed through softmax to obtain probability distribution which can be evaluated for cross-entropy loss. Finally, we evaluate the model for accuracy using the test data.

```
import numpy as np
import keras
from keras.datasets import mnist
```

```
from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.utils import to_categorical

# Size of image is 28x28x1 channel.
input_shape = (28, 28, 1)
batch_size = 64
# number of possible outcomes [0-9]
nclasses = 10
epochs = 3

# Fetch the train and test data.
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the image so that all pixel values
# are between -0.5 and +0.5.
x_train = (x_train / 255) - 0.5
x_test = (x_test / 255) - 0.5

# Reshape the train and test images to size 28x28x1.
x_train = x_train.reshape((x_train.shape[0], *input_shape))
x_test = x_test.reshape((x_test.shape[0], *input_shape))

# Define the CNN model with 2 convolution layer and
# 2 max pooling layer followed by a neural network
# with 1 hidden layer of size 128 nodes.
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
```

```
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(nclasses, activation='softmax'))

# Compile the model using Adam optimizer and use
# the cross entropy loss.
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model.
model.fit(x_train, to_categorical(y_train), epochs=epochs,
          batch_size=batch_size)

# Evaluate the model.
score = model.evaluate(x_test, to_categorical(y_test),
    verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

In the case of CNN, in comparison to the neural network example from the previous chapter, you will notice that we reached a high accuracy value in fewer epochs although each epoch took a little longer.

```
Epoch 1/3
60000/60000 [====] - 33s 555us/step - loss: 0.1683 -
accuracy: 0.9504
Epoch 2/3
60000/60000 [====] - 27s 444us/step - loss: 0.0493 -
accuracy: 0.9847
Epoch 3/3
60000/60000 [====] - 48s 792us/step - loss: 0.0331 -
accuracy: 0.9898
```

```
Test loss: 0.03353308427521261
Test accuracy: 0.9894000291824341
```

12.5 Summary

- CNN was originally developed as a mathematical model of vision. Hence they are well suited for solving computer vision problems.
- CNNs are created by composition of convolution and maxpooling layer followed by a classifier or regressor, which is typically a neural network.
- The parameters of the convolution layer are learned using the back-propagation process.

12.6 Exercises

1. What is the effect of increasing the number of convolution layers in a neural network?
2. Modify the above code and run it with the FashionMNIST data set available at https://github.com/zalandoresearch/fashion-mnist. This data set also has 10 categories such a trousers, shoes, etc.