Figure 35-15 is an example of a CNN architecture.

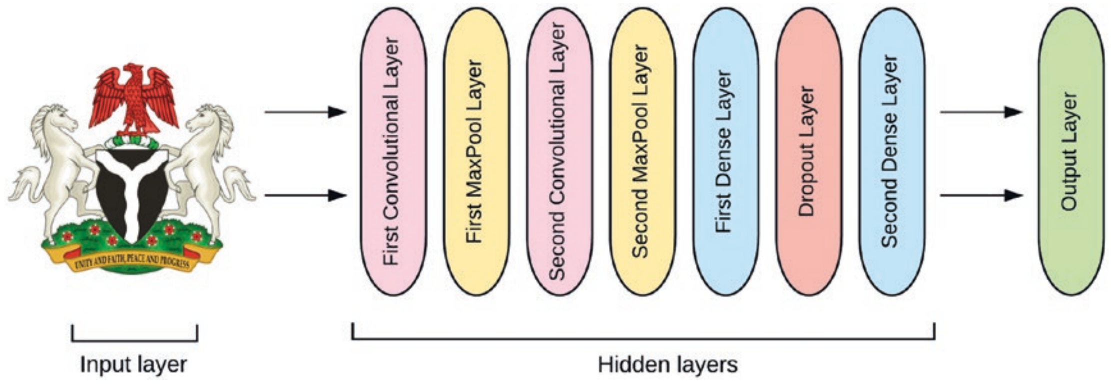


Figure 35-15. *CNN architecture*

CNN for Image Recognition with TensorFlow 2.0

In this example, we will build a convolutional neural network (CNN) to classify images from the CIFAR-10 dataset. CIFAR-10 is another standard image classification dataset to classify a colored 32 x 32 pixel image data into ten image classes, namely, airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The focus of this section is exclusively on using TensorFlow 2.0 methods to build a CNN image classifier.

The CNN model architecture implemented loosely mirrors the Krizhevsky's architecture, also known as AlexNet. The network architecture has the following layers:

- Convolution layer: kernel_size = [5 x 5]
- Convolution layer: kernel_size = [5 x 5]
- Batch normalization layer
- Convolution layer: kernel_size = [5 x 5]
- Max pooling: pool size = [2 x 2]
- Convolution layer: kernel_size = [5 x 5]
- Convolution layer: kernel_size = [5 x 5]
- Batch normalization layer
- Max pooling: pool size = [2 x 2]

- Convolution layer: kernel_size = [5 x 5]
- Convolution layer: kernel_size = [5 x 5]
- Convolution layer: kernel_size = [5 x 5]
- Max pooling: pool size = [2 x 2]
- Dropout layer
- Dense layer: units = [512]
- Dense layer: units = [256]
- Dropout layer
- Dense layer: units = [10]

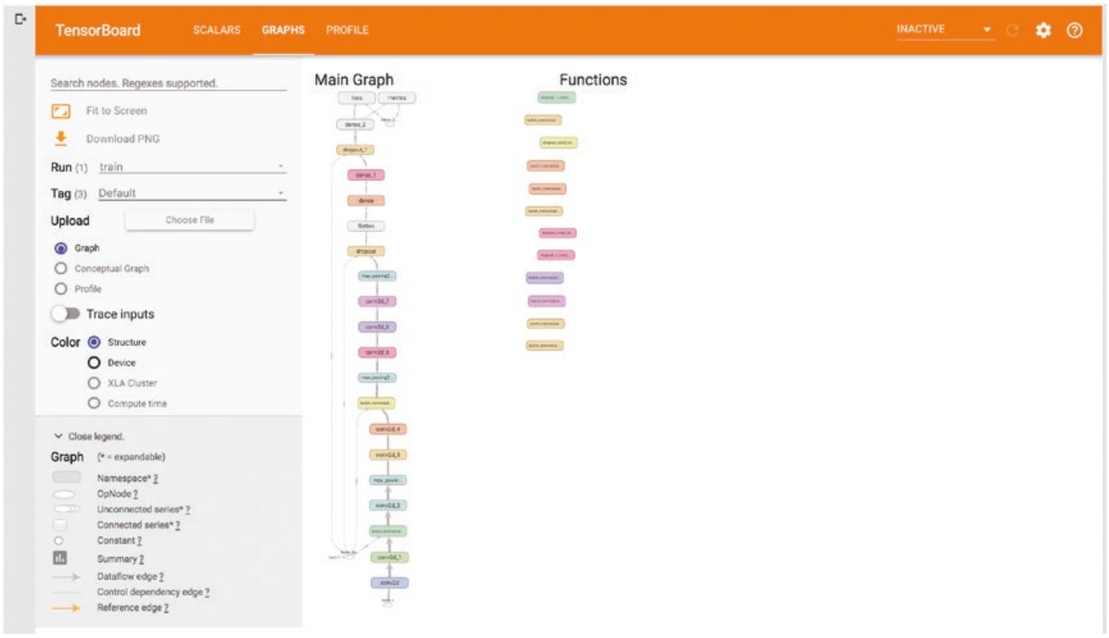
This CNN model has close to a million trainable variables as can be seen from the model summary when running **‘model.summary()’**. Training on a CPU will take an inordinate amount of time (about 1 hour and 30 minutes). For this code example, we will train on a GPU instance. If running the code on Google Colab, change the runtime type to GPU and install TensorFlow 2.0 with GPU package. The graph of the model in Tensorboard is shown in Figure 35-16.

```
# import TensorFlow 2.0 with GPU
!pip install -q tf-nightly-gpu-2.0-preview

# import packages
import tensorflow as tf

# confirm tensorflow can see GPU
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

# load the TensorBoard notebook extension
%load_ext tensorboard

# import dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

# change datatype to float
```

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

# scale the dataset from 0 -> 255 to 0 -> 1
x_train /= 255
x_test /= 255

# one-hot encode targets
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)

# parameters
batch_size = 100

# create dataset pipeline
train_ds = tf.data.Dataset.from_tensor_slices(
    (x_train, y_train)).shuffle(len(x_train)).repeat().batch(batch_size)
test_ds = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(batch_size)

# create the model
def model_fn():
    model_input = tf.keras.layers.Input(shape=(32, 32, 3))
    x = tf.keras.layers.Conv2D(64, (5, 5), padding='same',
    activation='relu')(model_input)
    x = tf.keras.layers.Conv2D(64, (5, 5), padding='same', activation='relu')(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Conv2D(64, (5, 5), padding='same', activation='relu')(x)
    x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=2,
    padding='same')(x)
    x = tf.keras.layers.Conv2D(64, (5, 5), padding='same', activation='relu')(x)
    x = tf.keras.layers.Conv2D(64, (5, 5), padding='same', activation='relu')(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=2, padding='same')(x)
    x = tf.keras.layers.Conv2D(32, (3, 3), padding='same', activation='relu')(x)
    x = tf.keras.layers.Conv2D(32, (3, 3), padding='same', activation='relu')(x)
    x = tf.keras.layers.Conv2D(32, (3, 3), padding='same', activation='relu')(x)
    x = tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=2,
    padding='same')(x)
    x = tf.keras.layers.Dropout(0.3)(x)
    x = tf.keras.layers.Flatten()(x)
    x = tf.keras.layers.Dense(512, activation='relu')(x)
    x = tf.keras.layers.Dense(256, activation='relu')(x)
    x = tf.keras.layers.Dropout(0.5)(x)
    output = tf.keras.layers.Dense(10, activation='softmax')(x)

    # the model
    model = tf.keras.Model(inputs=model_input, outputs=output)

    # compile the model
    model.compile(optimizer=tf.keras.optimizers.Nadam(),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])
    return model

# build the model
model = model_fn()

# print model summary
model.summary()

# tensorboard
tensorboard = tf.keras.callbacks.TensorBoard(log_dir='./tmp/logs_cifar10_
              keras',
              histogram_freq=0, write_graph=True,
               write_images=True)

# assign callback
callbacks = [tensorboard]

# train the model
history = model.fit(train_ds, epochs=10,
                    steps_per_epoch=500,
                    callbacks=callbacks)

# evaluate the model
score = model.evaluate(test_ds)
print('Test loss: {:.2f} \nTest accuracy: {:.2f}%'.format(score[0], score[1]*100))
```

```
'Output:'
Test loss: 0.74
Test accuracy: 80.05%
```

```
# execute the command to run TensorBoard
%tensorboard --logdir tmp/logs_cifar10_keras
```



Run (1) train
Tag (3) Default
● Graph
○ Conceptual Graph
○ Profile
Color ● Structure
○ Device
○ XLA Cluster
○ Compute time

Figure 35-16. *Tensorboard output of CIFAR-10 model graph*

In this chapter, we discussed the convolutional neural network (CNN) as an example of a deep neural network. We went through the design details in architecting a CNN and implemented a CNN model with TensorFlow 2.0. In the next chapter, we will examine another type of deep neural network called the recurrent neural network.

CHAPTER 36

Recurrent Neural Networks (RNNs)

Recurrent neural networks (RNNs) are another specialized scheme of neural network architectures. RNNs are developed to solve learning problems where information about the past (i.e., past instants/events) is directly linked to making future predictions. Such sequential examples play up frequently in many real-world tasks such as language modeling where the previous words in the sentence are used to determine what the next word will be. Also in stock market prediction, the last hour/day/week stock prices define the future stock movement. RNNs are particularly tuned for time series or sequential tasks.

In a sequential problem, there is a looping or feedback framework that connects the output of one sequence to the input of the next sequence. RNNs are ideal for processing 1-D sequential data, unlike the grid-like 2-D image data in convolutional neural networks.

This feedback framework enables the network to incorporate information from past sequences or from time-dependent datasets when making a prediction.
In this section, we will cover the broad conceptual overview of recurrent neural networks and in particular the Long Short-Term Memory RNN variant (LSTM) which is the state-of-the-art technique for various sequential problems such as image captioning, stock market prediction, machine translation, and text classification.

The Recurrent Neuron

The first building block of the RNN is the recurrent neuron (see Figure 36-1). The neurons of the recurrent network are entirely different from those of other neural network architectures. The key difference here is that the recurrent neuron maintains a memory or a state from past computations. It does this by taking as input the output of the previous instant y_{t-1} in addition to its current input at a particular instant x_t.

© Ekaba Bisong 2019
E. Bisong, *Building Machine Learning and Deep Learning Models on Google Cloud Platform*,
https://doi.org/10.1007/978-1-4842-4470-8_36