# 3
# Convolution Neural Network

In this chapter, we will cover the following topics:

- Downloading and configuring an image dataset
- Learning the architecture of a CNN classifier
- Using functions to initialize weights and biases
- Using functions to create a new convolution layer
- Using functions to flatten the densely connected layer
- Defining placeholder variables
- Creating the first convolution layer
- Creating the second convolution layer
- Flattening the second convolution layer
- Creating the first fully connected layer
- Applying dropout to the first fully connected layer
- Creating the second fully connected layer with dropout
- Applying softmax activation to obtain a predicted class
- Defining the cost function used for optimization
- Performing gradient descent cost optimization
- Executing the graph in a TensorFlow session
- Evaluating the performance on test data

# Introduction

**Convolution neural networks** (**CNN**) are a category of deep learning neural networks with a prominent role in building image recognition- and natural language processing-based classification models.

The CNN follows a similar architecture to LeNet, which was primarily designed to recognize characters such as numbers, zip codes, and so on. As against artificial neural networks, CNN have layers of neurons arranged in three-dimensional space (width, depth, and height). Each layer transforms a two-dimensional image into a three-dimensional input volume, which is then transformed into a three-dimensional output volume using neuron activation.

Primarily, CNNs are built using three main types of activation layers: convolution layer ReLU, pooling layer, and fully connected layer. The convolution layer is used to extract features (spatial relationship between pixels) from the input vector (of images) and stores them for further processing after computing a dot product with weights (and biases).

**Rectified Linear Unit** (**ReLU**) is then applied after convolution to introduce non-linearity in the operation.

This is an element-wise operation (such as a threshold function, sigmoid, and tanh) applied to each convolved feature map. Then, the pooling layer (operations such as max, mean, and sum) is used to downsize the dimensionality of each feature map ensuring minimum information loss. This operation of spatial size reduction is used to control overfitting and increase the robustness of the network to small distortions or transformations. The output of the pooling layer is then connected to a traditional multilayer perceptron (also called the fully connected layer). This perceptron uses activation functions such as softmax or SVM to build classifier-based CNN models.

The recipes in this chapter will focus on building a convolution neural network for image classification using Tensorflow in R. While the recipes will provide you with an overview of a typical CNN, we encourage you to adapt and modify the parameters according to your needs.

# Downloading and configuring an image dataset

In this chapter, we will use the CIFAR-10 dataset to build a convolution neural network for image classification. The CIFAR-10 dataset consists of 60,000 32 x 32 color images of 10 classes, with 6,000 images per class. These are further divided into five training batches and one test batch, each with 10,000 images.

The test batch contains exactly 1,000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5,000 images from each class. The ten outcome classes are airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The classes are completely mutually exclusive. In addition, the format of the dataset is as follows:

- The first column: The label with 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck
- The next 1,024 columns: Red pixels in the range of 0 to 255
- The next 1,024 columns: Green pixels in the range of 0 to 255
- The next 1,024 columns: Blue pixels in the range of 0 to 255

## Getting ready

For this recipe, you will require R with some packages installed such as `data.table` and `imager`.

## How to do it...

1. Start R (using Rstudio or Docker) and load the required packages.
2. Download the dataset (binary version) from `http://www.cs.toronto.edu/~kriz/cifar.html`manually or use the following function to download the data in the R environment. The function takes the working directory or the downloaded dataset's location path as an input parameter (`data_dir`):

```
# Function to download the binary file
download.cifar.data <- function(data_dir) {
dir.create(data_dir, showWarnings = FALSE)
setwd(data_dir)
```

```
if (!file.exists('cifar-10-binary.tar.gz')){
download.file(url='http://www.cs.toronto.edu/~kriz/cifar-10-binary.
tar.gz', destfile='cifar-10-binary.tar.gz', method='wget')
untar("cifar-10-binary.tar.gz") # Unzip files
file.remove("cifar-10-binary.tar.gz") # remove zip file
}
setwd("..")
}
# Download the data
download.cifar.data(data_dir="Cifar_10/")
```

3. Once the dataset is downloaded and untarred (or unzipped), read it in the R environment as train and test datasets. The function takes the filenames of the train and test batch datasets (`filenames`) and the number of images to retrieve per batch file (`num.images`) as input parameters:

```
# Function to read cifar data
read.cifar.data <- function(filenames,num.images){
images.rgb <- list()
images.lab <- list()
for (f in 1:length(filenames)) {
to.read <- file(paste("Cifar_10/",filenames[f], sep=""), "rb")
for(i in 1:num.images) {
l <- readBin(to.read, integer(), size=1, n=1, endian="big")
r <- as.integer(readBin(to.read, raw(), size=1, n=1024,
endian="big"))
g <- as.integer(readBin(to.read, raw(), size=1, n=1024,
endian="big"))
b <- as.integer(readBin(to.read, raw(), size=1, n=1024,
endian="big"))
index <- num.images * (f-1) + i
images.rgb[[index]] = data.frame(r, g, b)
images.lab[[index]] = l+1
}
close(to.read)
cat("completed :", filenames[f], "\n")
remove(l,r,g,b,f,i,index, to.read)
}
return(list("images.rgb"=images.rgb,"images.lab"=images.lab))
}
# Train dataset
cifar_train <- read.cifar.data(filenames =
c("data_batch_1.bin","data_batch_2.bin","data_batch_3.bin","data_ba
tch_4.bin", "data_batch_5.bin"))
images.rgb.train <- cifar_train$images.rgb
images.lab.train <- cifar_train$images.lab
rm(cifar_train)
```

```
# Test dataset
cifar_test <- read.cifar.data(filenames = c("test_batch.bin"))
images.rgb.test <- cifar_test$images.rgb
images.lab.test <- cifar_test$images.lab
rm(cifar_test)
```

4. The outcome of the earlier function is a list of red, green, and blue pixel dataframes for each image along with their labels. Then, flatten the data into a list of two dataframes (one for input and the other for output) using the following function, which takes two parameters--a list of input variables (`x_listdata`) and a list of output variables (`y_listdata`):

```
# Function to flatten the data
flat_data <- function(x_listdata,y_listdata){
# Flatten input x variables
x_listdata <- lapply(x_listdata,function(x){unlist(x)})
x_listdata <- do.call(rbind,x_listdata)
# Flatten outcome y variables
y_listdata <- lapply(y_listdata,function(x){a=c(rep(0,10)); a[x]=1;
return(a)})
y_listdata <- do.call(rbind,y_listdata)
# Return flattened x and y variables
return(list("images"=x_listdata, "labels"=y_listdata))
}
# Generate flattened train and test datasets
train_data <- flat_data(x_listdata = images.rgb.train, y_listdata =
images.lab.train)
test_data <- flat_data(x_listdata = images.rgb.test, y_listdata =
images.lab.test)
```

5. Once the list of input and output train and test dataframes is ready, perform sanity checks by plotting the images along with their labels. The function requires two mandatory parameters (`index`: image row number and `images.rgb`: flattened input dataset) and one optional parameter (`images.lab`: flattened output dataset):

```
labels <- read.table("Cifar_10/batches.meta.txt")
# function to run sanity check on photos & labels import
drawImage <- function(index, images.rgb, images.lab=NULL) {
require(imager)
# Testing the parsing: Convert each color layer into a matrix,
# combine into an rgb object, and display as a plot
img <- images.rgb[[index]]
img.r.mat <- as.cimg(matrix(img$r, ncol=32, byrow = FALSE))
img.g.mat <- as.cimg(matrix(img$g, ncol=32, byrow = FALSE)
img.b.mat <- as.cimg(matrix(img$b, ncol=32, byrow = FALSE))
```

```
img.col.mat <- imappend(list(img.r.mat,img.g.mat,img.b.mat),"c")
#Bind the three channels into one image
# Extract the label
if(!is.null(images.lab)){
lab = labels[[1]][images.lab[[index]]]
}
# Plot and output label
plot(img.col.mat,main=paste0(lab,":32x32 size",sep=" "),xaxt="n")
axis(side=1, xaxp=c(10, 50, 4), las=1)
return(list("Image label" =lab,"Image description" =img.col.mat))
}
# Draw a random image along with its label and description from
train dataset
drawImage(sample(1:50000, size=1), images.rgb.train,
images.lab.train)
```

6. Now transform the input data using the min-max standardization technique. The `preProcess` function from the package can be used for normalization. The `"range"` option of the method performs min-max normalization as follows:

```
# Function to normalize data
Require(caret)
normalizeObj<-preProcess(train_data$images, method="range")
train_data$images<-predict(normalizeObj, train_data$images)
test_data$images <- predict(normalizeObj, test_data$images)
```

# How it works...

Let's take a look at what we did in the earlier recipe. In step 2, we downloaded the CIFAR-10 dataset from the link mentioned in case it is not present in the given link or working directory. In step 3, the unzipped files are loaded in the R environment as train and test datasets. The train dataset has a list of 50,000 images and the test dataset has a list of 10,000 images along with their labels. Then, in step 4, the train and test datasets are flattened into a list of two dataframes: one with input variables (or images) of length 3,072 (1,024 of red, 1,024 of green, and 1,024 of blue) and the other with output variables (or labels) of length 10 (binary for each class). In step 5, we perform sanity checks for the created train and test datasets by generating plots. The following figure shows a set of six train images along with their labels. Finally, in step 6, the input data is transformed using the min-max standardization technique. An example of categories from the CIFAR-10 dataset is shown in the following figure:


## See also

*Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009* (`http://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf`). This is also the reference for this section.

# Learning the architecture of a CNN classifier

The CNN classifier covered in this chapter has two convolution layers followed by two fully connected layers in the end, in which the last layer acts as a classifier using the softmax `activation` function.

## Getting ready

The recipe requires the CIFAR-10 dataset. Thus, the CIFAR-10 dataset should be downloaded and loaded into the R environment. Also, images are of size 32 x 32 pixels.

# How to do it...

Let's define the configuration of the CNN classifier as follows:

1. Each input image (CIFAR-10) is of size 32 x 32 pixels and can be labeled one among 10 classes:

```
# CIFAR images are 32 x 32 pixels.
img_width  = 32L
img_height = 32L

# Tuple with height and width of images used to reshape arrays.
img_shape = c(img_width, img_height)
# Number of classes, one class for each of 10 images
num_classes = 10L
```

2. The images of the CIFAR-10 dataset have three channels (red, green, and blue):

```
# Number of color channels for the images: 3 channel for red, blue,
green scales.
num_channels = 3L
```

3. The images are stored in one-dimensional arrays of the following length (`img_size_flat`):

```
# Images are stored in one-dimensional arrays of length.
img_size_flat = img_width * img_height * num_channels
```

4. In the first convolution layer, the size (width x height) of the convolution filter is 5 x 5 pixels (`filter_size1`) and the depth (or number) of convolution filter is `64` (`num_filters1`):

```
# Convolutional Layer 1.
filter_size1 = 5L
num_filters1 = 64L
```

5. In the second convolution layer, the size and depth of the convolution filter is the same as the first convolution layer:

```
# Convolutional Layer 2.
filter_size2 = 5L
num_filters2 = 64L
```

6. Similarly, the output of the first fully connected layer is the same as the input of the second fully connected layer:

```
# Fully-connected layer.
fc_size = 1024L
```

# How it works...

The dimensions and characteristics of an input image are shown in steps 1 and 2, respectively. Every input image is further processed in a convolution layer using a set of filters as defined in steps 4 and 5. The first convolution layer results in a set of 64 images (one for each set filter). In addition, the resolution of these images are also reduced to half (because of 2 x 2 max pooling); namely, from 32 x 32 pixels to 16 x 16 pixels.

The second convolution layer will input these 64 images and provide an output of new 64 images with further reduced resolutions. The updated resolution is now 8 x 8 pixels (again due to 2 x 2 max pooling). In the second convolution layer, a total of 64 x 64 = 4,096 filters are created, which are then further convoluted into 64 output images (or channels). Remember that these 64 images of 8 x 8 resolution correspond to a single input image.

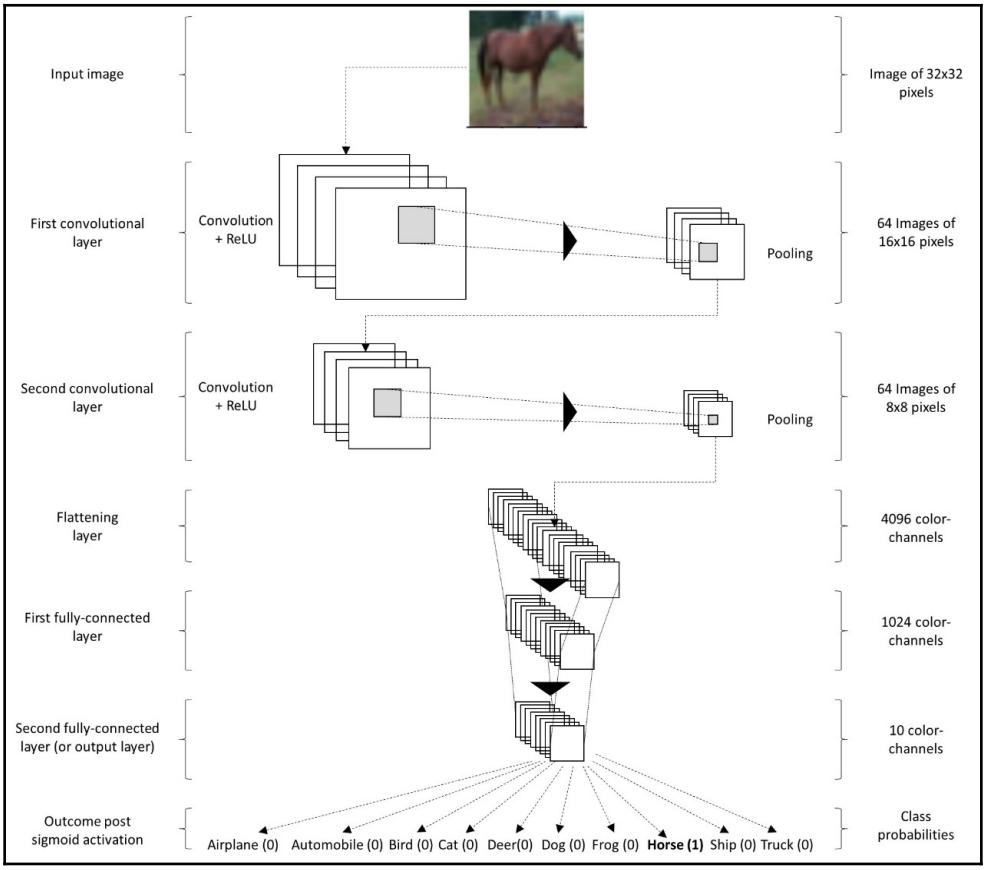
Further, these 64 output images of 8 x 8 pixels are flattened into a single vector of length 4,096 (8 x 8 x 64), as defined in step 3, and are used as an input to a fully connected layer of a given set of neurons, as defined in step 6. The vector of 4,096 elements is then fed into the first fully connected layer of 1,024 neurons. The output neurons are again fed into a second fully connected layer of 10 neurons (equal to `num_classes`). These 10 neurons represent each of the class labels, which are then used to determine the (final) class of the image.

First, the weights of the convolution and fully connected layers are randomly initialized till the classification stage (the end of CNN graph). Here, the classification error is computed based on the true class and the predicted class (also called cross entropy).

Then, the optimizer backpropagates the error through the convolution network using the chain rule of differentiation, post which the weights of the layers (or filters) are updated such that the error is minimized. This entire cycle of one forward and backward propagation is called one iteration. Thousands of such iterations are performed till the classification error is reduced to a sufficiently low value.

Generally, these iterations are performed using a batch of images instead of a single image to increase the efficiency of computation.

The following image represents the convolution network designed in this chapter:




# Using functions to initialize weights and biases

Weights and biases form an integral part of any deep neural network optimization and here we define a couple of functions to automate these initializations. It is a good practice to initialize weights with small noise to break symmetry and prevent zero gradients. Additionally, a small positive initial bias would avoid inactivated neurons, suitable for ReLU activation neurons.

# Getting ready

Weights and biases are model coefficients which need to be initialized before model compilation. This steps require the `shape` parameter to be determined based on input dataset.

# How to do it...

1. The following function is used to return randomly initialized weights:

```
# Weight Initialization
weight_variable <- function(shape) {
initial <- tf$truncated_normal(shape, stddev=0.1)
tf$Variable(initial)
}
```

2. The following function is used to return constant biases:

```
bias_variable <- function(shape) {
initial <- tf$constant(0.1, shape=shape)
tf$Variable(initial)
}
```

# How it works...

These functions return TensorFlow variables that are later used as part of a Tensorflow graph. The `shape` is defined as a list of attributes defining a filter in the convolution layer, which is covered in the next recipe. The weights are randomly initialized with a standard deviation equal to `0.1` and biases are initialized with a constant value of `0.1`.

# Using functions to create a new convolution layer

Creating a convolution layer is the primary step in a CNN TensorFlow computational graph. This function is primarily used to define the mathematical formulas in the TensorFlow graph, which is later used in actual computation during optimization.

# Getting ready

The input dataset is defined and loaded. The `create_conv_layer` function presented in the recipe takes the following five input parameters and needs to be defined while setting-up a convolution layer:

1. `Input`: This is a four-dimensional tensor (or a list) that comprises a number of (input) images, the height of each image (here 32L), the width of each image (here 32L), and the number of channels of each image (here 3L : red, blue, and green).
2. `Num_input_channels`: This is defined as the number of color channels in the case of the first convolution layer or the number of filter channels in the case of subsequent convolution layers.
3. `Filter_size`: This is defined as the width and height of each filter in the convolution layer. Here, the filter is assumed to be a square.
4. `Num_filters`: This is defined as the number of filters in a given convolution layer.
5. `Use_pooling`: This is a binary variable that is used perform 2 x 2 max pooling.

# How to do it...

1. Run the following function to create a new convolution layer:

```
# Create a new convolution layer
create_conv_layer <- function(input,
num_input_channels,
filter_size,
num_filters,
use_pooling=True)
{
# Shape of the filter-weights for the convolution.
shape1 = shape(filter_size, filter_size, num_input_channels,
num_filters)
# Create new weights
weights = weight_variable(shape=shape1)
# Create new biases
biases = bias_variable(shape=shape(num_filters))
# Create the TensorFlow operation for convolution.
layer = tf$nn$conv2d(input=input,
filter=weights,
strides=shape(1L, 1L, 1L ,1L),
padding="SAME")
# Add the biases to the results of the convolution.
```

```
layer = layer + biases
# Use pooling (binary flag) to reduce the image resolution
if(use_pooling){
layer = tf$nn$max_pool(value=layer,
ksize=shape(1L, 2L, 2L, 1L),
strides=shape(1L, 2L, 2L, 1L),
padding='SAME')
}
# Add non-linearity using Rectified Linear Unit (ReLU).
layer = tf$nn$relu(layer)
# Retrun resulting layer and updated weights
return(list("layer" = layer, "weights" = weights))
}
```

2. Run the following function to generate plots of convolution layers:

```
drawImage_conv <- function(index, images.bw,
images.lab=NULL,par_imgs=8) {
require(imager)
img <- images.bw[index,,,]
n_images <- dim(img)[3]
par(mfrow=c(par_imgs,par_imgs), oma=c(0,0,0,0),
mai=c(0.05,0.05,0.05,0.05),ann=FALSE,ask=FALSE)
for(i in 1:n_images){
img.bwmat <- as.cimg(img[,,i])
# Extract the label
if(!is.null(images.lab)){
lab = labels[[1]][images.lab[[index]]]
}
# Plot and output label
plot(img.bwmat,axes=FALSE,ann=FALSE)
}
par(mfrow=c(1,1))
}
```

3. Run the following function to generate plots of convolution layer weights:

```
drawImage_conv_weights <- function(weights_conv, par_imgs=8) {
require(imager)
n_images <- dim(weights_conv)[4]
par(mfrow=c(par_imgs,par_imgs), oma=c(0,0,0,0),
mai=c(0.05,0.05,0.05,0.05),ann=FALSE,ask=FALSE)
for(i in 1:n_images){
img.r.mat <- as.cimg(weights_conv[,,1,i])
img.g.mat <- as.cimg(weights_conv[,,2,i])
img.b.mat <- as.cimg(weights_conv[,,3,i])
img.col.mat <- imappend(list(img.r.mat,img.g.mat,img.b.mat),"c")
#Bind the three channels into one image
```

```
# Plot and output label
plot(img.col.mat,axes=FALSE,ann=FALSE)
}
par(mfrow=c(1,1))
}
```

# How it works...

The function begins with creating a shape tensor; namely, the list of four integers that are the width of a filter, the height of a filter, the number of input channels, and the number of given filters. Using this shape tensor, initialize a tensor of new weights with the defined shape and create a tensor a new (constant) biases, one for each filter.

Once the necessary weights and biases are initialized, create a TensorFlow operation for convolution using the `tf$nn$conv2d` function. In our current setup, the strides are set to 1 in all four dimensions and padding is set to `SAME`. The first and last are set to 1 by default, but the middle two can factor in higher strides. A stride is the number of pixels by which we allow the filter matrix to slide over the input (image) matrix.

A stride of 3 would mean three pixel jumps across the *x* or *y* axis for each filter slide. Smaller strides would produce larger feature maps, thereby requiring higher computation for convergence. As the padding is set to `SAME`, the input (image) matrix is padded with zeros around the border so that we can apply the filter to border elements of the input matrix. Using this feature, we can control the size of the output matrix (or feature maps) to be the same as the input matrix.

On convolution, the bias values are added to each filter channel followed by pooling to prevent overfitting. In the current setup, 2 x 2 max-pooling (using `tf$nn$max_pool`) is performed to downsize the image resolution. Here, we consider 2 x 2 (`ksize`)-sized windows and select the largest value in each window. These windows stride by two pixels (`strides`) either in the x or y direction.

On pooling, we add non-linearity to the layer using the ReLU activation function (`tf$nn$relu`). In ReLU, each pixel is triggered in the filter and all negative pixel values are replaced with zero using the `max(x,0)` function, where *x* is a pixel value. Generally, ReLU activation is performed before pooling. However, as we are using max-pooling, it doesn't necessarily impact the outcome as such because `relu(max_pool(x))` is equivalent to `max_pool(relu(x))`. Thus, by applying ReLU post pooling, we can save a lot of ReLU operations (~75%).

Finally, the function returns a list of convoluted layers and their corresponding weights. The convoluted layer is a four-dimensional tensor with the following attributes:

- Number of (input) images, the same as `input`
- Height of each image (reduced to half in the case of 2 x 2 max-pooling)
- Width of each image (reduced to half in the case of 2 x 2 max-pooling)
- Number of channels produced, one for each convolution filter

# Using functions to create a new convolution layer

The four-dimensional outcome of a newly created convolution layer is flattened to a two-dimensional layer such that it can be used as an input to a fully connected multilayered perceptron.

## Getting ready

The recipe explains how to flatten a convolution layer before building the deep learning model. The input to the given function ( `flatten_conv_layer`) is a four-dimensional convolution layer that is defined based on previous layer.

## How to do it...

1. Run the following function to flatten the convolution layer:

```
flatten_conv_layer <- function(layer){
# Extract the shape of the input layer
layer_shape = layer$get_shape()
# Calculate the number of features as img_height * img_width *
num_channels
num_features =
prod(c(layer_shape$as_list()[[2]],layer_shape$as_list()[[3]],layer_
shape$as_list()[[4]]))
# Reshape the layer to [num_images, num_features].
layer_flat = tf$reshape(layer, shape(-1, num_features))
# Return both the flattened layer and the number of features.
return(list("layer_flat"=layer_flat, "num_features"=num_features))
}
```

## How it works...

The function begins with extracting the shape of the given input layer. As stated in previous recipes, the shape of the input layer comprises four integers: image number, image height, image width, and the number of color channels in the image. The number of features (`num_features`) is then evaluated using a dot-product of image height, image weight, and number of color channels.

Then, the layer is flattened or reshaped into a two-dimensional tensor (using `tf$reshape`). The first dimension is set to -1 (which is equal to the total number of images) and the second dimension is the number of features.

Finally, the function returns a list of flattened layers along with the total number of (input) features.

# Using functions to flatten the densely connected layer

The CNN generally ends with a fully connected multilayered perceptron using softmax activation in the output layer. Here, each neuron in the previous convoluted-flattened layer is connected to every neuron in the next (fully connected) layer.

The key purpose of the fully convoluted layer is to use the features generated in the convolution and pooling stage to classify the given input image into various outcome classes (here, 10L). It also helps in learning the non-linear combinations of these features to define the outcome classes.

In this chapter, we use two fully connected layers for optimization. This function is primarily used to define the mathematical formulas in the TensorFlow graph, which is later used in actual computation during optimization.

## Getting ready

The (`create_fc_layer`) function takes four input parameters, which are as follows:

- `Input`: This is similar to the input of the new convolution layer function

- `Num_inputs`: This is the number of input features generated post flattening the convoluted layer
- `Num_outputs`: This is the number of output neurons fully connected with the input neurons
- `Use_relu`: This takes the binary flag that is set to `FALSE` only in the case of the final fully connected layer

# How to do it...

1. Run the following function to create a new fully connected layer:

```
# Create a new fully connected layer
create_fc_layer <- function(input,
num_inputs,
num_outputs,
use_relu=True)
{
# Create new weights and biases.
weights = weight_variable(shape=shape(num_inputs, num_outputs))
biases = bias_variable(shape=shape(num_outputs))
# Perform matrix multiplication of input layer with weights and
then add biases
layer = tf$matmul(input, weights) + biases
# Use ReLU?
if(use_relu){
layer = tf$nn$relu(layer)
}
return(layer)
}
```

# How it works...

The function begins with initialing new weights and biases. Then, perform matrix multiplication of the input layer with initialized weights and add relevant biases.

If, the fully connected layer is not the final layer of the CNN TensorFlow graph, ReLU non-linear activation can be performed. Finally, the fully connected layer is returned.

# Defining placeholder variables

In this recipe, let's define the placeholder variables that serve as input to the modules in a TensorFlow computational graph. These are typically multidimensional arrays or matrices in the form of tensors.

# Getting ready

The data type of placeholder variables is set to float32 (`tf$float32`) and the shape is set to a two-dimensional tensor.

# How to do it...

1. Create an input placeholder variable:

```
x = tf$placeholder(tf$float32, shape=shape(NULL, img_size_flat),
name='x')
```

   The NULL value in the placeholder allows us to pass non-deterministic arrays size.

2. Reshape the input placeholder `x` into a four-dimensional tensor:

```
x_image = tf$reshape(x, shape(-1L, img_size, img_size,
num_channels))
```

3. Create an output placeholder variable:

```
y_true = tf$placeholder(tf$float32, shape=shape(NULL, num_classes),
name='y_true')
```

4. Get the (`true`) classes of the output using argmax:

```
y_true_cls = tf$argmax(y_true, dimension=1L)
```

## How it works...

In step 1, we define an input placeholder variable. The dimensions of the shape tensor are `NULL` and `img_size_flat`. The former is set to hold any number of images (as rows) and the latter defines the length of input features for each image (as columns). In step 2, the input two-dimensional tensor is reshaped into a four-dimensional tensor, which can be served as input convolution layers. The four dimensions are as follows:

- The first defines the number of input images (currently set to -1)
- The second defines the height of each image (equivalent to image size 32L)
- The third defines the width of each image (equivalent to image size, again 32L)
- The fourth defines the number of color channels in each image (here 3L)

In step 3, we define an output placeholder variable to hold true classes or labels of the images in `x`. The dimensions of the shape tensor are `NULL` and `num_classes`. The former is set to hold any number of images (as rows) and the latter defines the true class of each image as a binary vector of length `num_classes` (as columns). In our scenario, we have 10 classes. In step 4, we compress the two-dimensional output placeholder into a one-dimensional tensor of class numbers ranging from 1 to 10.

# Creating the first convolution layer

In this recipe, let's create the first convolution layer.

## Getting ready

The following are the inputs to the function `create_conv_layer` defined in the recipe *Using functions to create a new convolution layer*.

- `Input`: This is a four-dimensional reshaped input placeholder variable: `x_image`
- `Num_input_channels`: This is the number of color channels, namely `num_channels`
- `Filter_size`: This is the height and width of the filter layer `filter_size1`
- `Num_filters`: This is the depth of the filter layer, namely `num_filters1`
- `Use_pooling`: This is the binary flag set to `TRUE`

# How to do it...

1. Run the `create_conv_layer` function with the preceding input parameters:

```
# Convolutional Layer 1
conv1 <- create_conv_layer(input=x_image,
num_input_channels=num_channels,
filter_size=filter_size1,
num_filters=num_filters1,
use_pooling=TRUE)
```

2. Extract the `layers` of the first convolution layer:

```
layer_conv1 <- conv1$layer
conv1_images <- conv1$layer$eval(feed_dict = dict(x =
train_data$images, y_true = train_data$labels))
```

3. Extract the final `weights` of the first convolution layer:

```
weights_conv1 <- conv1$weights
weights_conv1 <- weights_conv1$eval(session=sess)
```

4. Generate the first convolution layer plots:

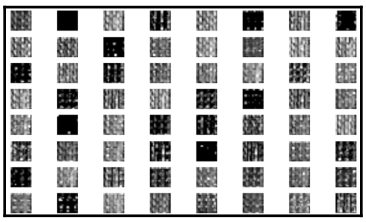
```
drawImage_conv(sample(1:50000, size=1), images.bw = conv1_images,
images.lab=images.lab.train)
```

5. Generate the first convolution layer weight plots:

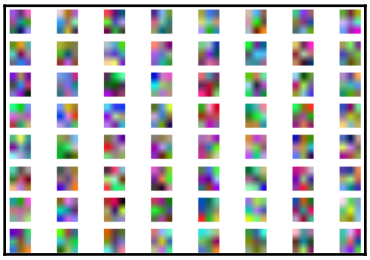
```
drawImage_conv_weights(weights_conv1)
```

# How it works...

In steps 1 and 2, we create a first convolution layer of four-dimensions. The first dimension (?) represents any number of input images, the second and third dimensions represent the height (16 pixels) and width (16 pixels) of each convoluted image, and the fourth dimension represents the number of channels (64) produced--one for each convoluted filter. In steps 3 and 5, we extract the final weights of the convolution layer, as shown in the following screenshot:



In step 4, we plot the output of the first convolution layer, as shown in the following screenshot:



# Creating the second convolution layer

In this recipe, let's create the second convolution layer.

# Getting ready

The following are the inputs to the function `create_conv_layer` defined in the recipe *Using functions to create a new convolution layer*.

- `Input`: This is the four-dimensional output of the first convoluted layer; that is, `layer_conv1`
- `Num_input_channels`: This is the number of filters (or depth) in the first convoluted layer, `num_filters1`
- `Filter_size`: This is the height and width of the filter layer; namely, `filter_size2`
- `Num_filters`: This is the depth of the filter layer, `num_filters2`
- `Use_pooling`: This is the binary flag set to `TRUE`

# How to do it...

1. Run the `create_conv_layer` function with the preceding input parameters:

```
# Convolutional Layer 2
conv2 <- create_conv_layer(input=layer_conv1,
num_input_channels=num_filters1,
filter_size=filter_size2,
num_filters=num_filters2,
use_pooling=TRUE)
```

2. Extract the layers of the second convolution layer:

```
layer_conv2 <- conv2$layer
conv2_images <- conv2$layer$eval(feed_dict = dict(x =
train_data$images, y_true = train_data$labels))
```

3. Extract the final weights of the second convolution layer:

```
weights_conv2 <- conv2$weights
weights_conv2 <- weights_conv2$eval(session=sess)
```

4. Generate the second convolution layer plots:

```
drawImage_conv(sample(1:50000, size=1), images.bw = conv2_images,
images.lab=images.lab.train)
```

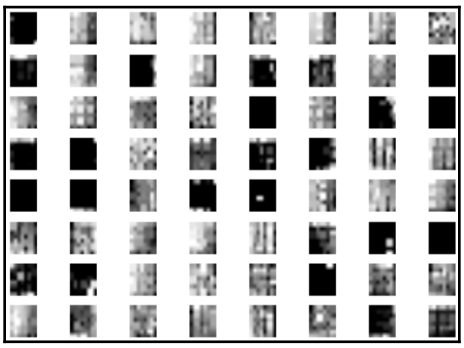
5. Generate the second convolution layer weight plots:

```
drawImage_conv_weights(weights_conv2)
```

# How it works...

In steps 1 and 2, we create a second convolution layer of four dimensions. The first dimension (?) represents any number of input images, the second and third dimensions represent the height (8 pixels) and width (8 pixels) of each convoluted image, and the fourth dimension represents the number of channels (64) produced, one for each convoluted filter.

In steps 3 and 5, we extract the final weights of the convolution layer, as shown in the following screenshot:



In step 4, we plot the output of the second convolution layer, as shown in the following screenshot:

| | predicted | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| actual | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 582 | 36 | 50 | 28 | 55 | 22 | 22 | 29 | 117 | 59 |
| 2 | 46 | 598 | 9 | 30 | 24 | 24 | 13 | 24 | 38 | 194 |
| 3 | 62 | 18 | 348 | 100 | 177 | 96 | 89 | 63 | 20 | 27 |
| 4 | 21 | 25 | 80 | 368 | 88 | 211 | 83 | 66 | 21 | 37 |
| 5 | 29 | 17 | 106 | 80 | 439 | 74 | 119 | 103 | 19 | 14 |
| 6 | 17 | 17 | 80 | 211 | 70 | 416 | 65 | 89 | 12 | 23 |
| 7 | 5 | 23 | 66 | 87 | 89 | 64 | 594 | 41 | 6 | 25 |
| 8 | 23 | 32 | 49 | 89 | 106 | 82 | 41 | 520 | 11 | 47 |
| 9 | 108 | 67 | 14 | 29 | 37 | 17 | 11 | 19 | 632 | 66 |
| 10 | 41 | 151 | 12 | 48 | 16 | 27 | 24 | 56 | 63 | 562 |

# Flattening the second convolution layer

In this recipe, let's flatten the second convolution layer that we created.

## Getting ready

The following is the input to the function defined in the recipe Creating the second convolution layer, `flatten_conv_layer`:

- `Layer`: This is the output of the second convolution layer, `layer_conv2`

# How to do it...

1. Run the `flatten_conv_layer` function with the preceding input parameter:

```
flatten_lay <- flatten_conv_layer(layer_conv2)
```

2. Extract the flattened layer:

```
layer_flat <- flatten_lay$layer_flat
```

3. Extract the number of (input) features generated for each image:

```
num_features <- flatten_lay$num_features
```

# How it works...

Prior to connecting the output of the (second) convolution layer with a fully connected network, in step 1, we reshape the four-dimensional convolution layer into a two-dimensional tensor. The first dimension (?) represents any number of input images (as rows) and the second dimension represents the flattened vector of features generated for each image of length 4,096; that is, 8 x 8 x 64 (as columns). Steps 2 and 3 validate the dimensions of the reshaped layers and input features.

# Creating the first fully connected layer

In this recipe, let's create the first fully connected layer.

# Getting ready

The following are the inputs to the function defined in the recipe *Using functions to flatten the densely connected layer*, `create_fc_layer`:

- `Input`: This is the flattened convolution layer; that is, `layer_flat`
- `Num_inputs`: This is the number of features created post flattening, `num_features`
- `Num_outputs`: This is the number of fully connected neurons output, `fc_size`
- `Use_relu`: This is the binary flag set to `TRUE` to incorporate non-linearity in the tensor

# How to do it...

1. Run the `create_fc_layer` function with the preceding input parameters:

```
layer_fc1 = create_fc_layer(input=layer_flat,
num_inputs=num_features,
num_outputs=fc_size,
use_relu=TRUE)
```

# How it works...

Here, we create a fully connected layer that returns a two-dimensional tensor. The first dimension (?) represents any number of (input) images and the second dimension represents the number of output neurons (here, 1,024).

# Applying dropout to the first fully connected layer

In this recipe, let's apply dropout to the output of the fully connected layer to reduce the chance of overfitting. The dropout step involves removing some neurons randomly during the learning process.

# Getting ready

The dropout is connected to the output of the layer. Thus, model initial structure is set up and loaded. For example, in dropout current layer `layer_fc1` is defined, on which dropout is applied.

# How to do it...

1. Create a placeholder for dropout that can take probability as an input:

```
keep_prob <- tf$placeholder(tf$float32)
```

2. Use TensorFlow's dropout function to handle the scaling and masking of neuron outputs:

```
layer_fc1_drop <- tf$nn$dropout(layer_fc1, keep_prob)
```

## How it works...

In steps 1 and 2, we can drop (or mask) out the output neurons based on the input probability (or percentage). The dropout is generally allowed during training and can be turned off (by assigning probability as `1` or `NULL`) during testing.

# Creating the second fully connected layer with dropout

In this recipe, let's create the second fully connected layer along with dropout.

## Getting ready

The following are the inputs to the function defined in the recipe *Using functions to flatten the densely connected layer*, `create_fc_layer`:

- `Input`: This is the output of the first fully connected layer; that is, `layer_fc1`
- `Num_inputs`: This is the number of features in the output of the first fully connected layer, `fc_size`
- `Num_outputs`: This is the number of the fully connected neurons output (equal to the number of labels, `num_classes` )
- `Use_relu`: This is the binary flag set to `FALSE`

# How to do it...

1. Run the `create_fc_layer` function with the preceding input parameters:

```
layer_fc2 = create_fc_layer(input=layer_fc1_drop,
num_inputs=fc_size,
num_outputs=num_classes,
use_relu=FALSE)
```

2. Use TensorFlow's dropout function to handle the scaling and masking of neuron outputs:

```
layer_fc2_drop <- tf$nn$dropout(layer_fc2, keep_prob)
```

# How it works...

In step 1, we create a fully connected layer that returns a two-dimensional tensor. The first dimension (?) represents any number of (input) images and the second dimension represents the number of output neurons (here, 10 class labels). In step 2, we provide the option for dropout primarily used during the training of the network.

# Applying softmax activation to obtain a predicted class

In this recipe, we will normalize the outputs of the second fully connected layer using softmax activation such that each class has a (probability) value restricted between 0 and 1, and all the values across 10 classes add up to 1.

# Getting ready

The activation function is applied at the end of the pipeline on predictions generated by the deep learning model. Before executing this step, all steps in the pipeline need to be executed. The recipe requires the TensorFlow library.

## How to do it...

1. Run the `softmax` activation function on the output of the second fully connected layer:

```
y_pred = tf$nn$softmax(layer_fc2_drop)
```

2. Use the `argmax` function to determine the class number of the label. It is the index of the class with the largest (probability) value:

```
y_pred_cls = tf$argmax(y_pred, dimension=1L)
```

# Defining the cost function used for optimization

The cost function is primarily used to evaluate the current performance of the model by comparing the true class labels (`y_true_cls`) with the predicted class labels (`y_pred_cls`). Based on the current performance, the optimizer then fine-tunes the network parameters, such as weights and biases, to further improve its performance.

## Getting ready

The cost function definition is critical as it will decide optimization criteria. The cost function definition will require true classes and predicted classes to do comparison. The objective function used in this recipe is cross entropy, used in multi-classification problems.

## How to do it...

1. Evaluate the current performance of each image using the cross entropy function in TensorFlow. As the cross entropy function in TensorFlow internally applies softmax normalization, we provide the output of the fully connected layer post dropout (`layer_fc2_drop`) as an input along with true labels (`y_true`):

```
cross_entropy =
tf$nn$softmax_cross_entropy_with_logits(logits=layer_fc2_drop,
labels=y_true)
```

In the current cost function, softmax activation function is embedded thus the activation function is not required to be defined separately.

2. Calculate the average of the cross entropy, which needs to be minimized using an optimizer:

```
cost = tf$reduce_mean(cross_entropy)
```

## How it works...

In step 1, we define a cross entropy to evaluate the performance of classification. Based on the exact match between the true and predicted labels, the cross entropy function returns a value that is positive and follows a continuous distribution. As zero cross entropy ensures a full match, optimizers tend to minimize the cross entropy toward the value zero by updating the network parameters such as weights and biases. The cross entropy function returns a value for each individual image that needs to be further compressed into a single scalar value, which can be used in an optimizer. Hence, in step 2, we calculate a simple average of the cross entropy output and store it as *cost*.

# Performing gradient descent cost optimization

In this recipe, let's define an optimizer that can minimize the cost. Post optimization, check for CNN performance.

## Getting ready

The optimizer definition will require the `cost` recipe to be defined as it goes as input to the optimizer.

## How to do it...

1. Run an Adam optimizer with the objective of minimizing the cost for a given `learning_rate`:

```
optimizer =
tf$train$AdamOptimizer(learning_rate=1e-4)$minimize(cost)
```

2. Extract the number of `correct_predictions` and calculate the mean percentage accuracy:

```
correct_prediction = tf$equal(y_pred_cls, y_true_cls)
accuracy = tf$reduce_mean(tf$cast(correct_prediction, tf$float32))
```

# Executing the graph in a TensorFlow session

Until now, we have only created tensor objects and added them to a TensorFlow graph for later execution. In this recipe, we will learn how to create a TensorFlow session that can be used to execute (or run) the TensorFlow graph.

## Getting ready

Before we run the graph, we should have TensorFlow installed and loaded in R. The installation details can be found in `Chapter 1`, *Getting Started*.

## How to do it...

1. Load the `tensorflow` library and import the `numpy` package:

```
library(tensorflow)
np <- import("numpy")
```

2. Reset or remove any existing `default_graph`:

```
tf$reset_default_graph()
```

3. Start an `InteractiveSession`:

```
sess <- tf$InteractiveSession()
```

4. Initialize the `global_variables`:

```
sess$run(tf$global_variables_initializer())
```

5. Run iterations to perform optimization (`training`):

```
# Train the model
train_batch_size = 128L
for (i in 1:100) {
spls <- sample(1:dim(train_data$images)[1],train_batch_size)
if (i %% 10 == 0) {
train_accuracy <- accuracy$eval(feed_dict = dict(
x = train_data$images[spls,], y_true = train_data$labels[spls,],
keep_prob = 1.0))
cat(sprintf("step %d, training accuracy %g\n", i, train_accuracy))
}
optimizer$run(feed_dict = dict(
x = train_data$images[spls,], y_true = train_data$labels[spls,],
keep_prob = 0.5))
}
```

6. Evaluate the performance of the trained model on test data:

```
# Test the model
test_accuracy <- accuracy$eval(feed_dict = dict(
x = test_data$images, y_true = test_data$labels, keep_prob = 1.0))
cat(sprintf("test accuracy %g", test_accuracy))
```

# How it works...

Steps 1 through 4 are, in a way, the default way to launch a new TensorFlow session. In step 4, the variables of weights and biases are initialized, which is mandatory before their optimization. Step 5 is primarily to execute the TensorFlow session for optimization. As we have a large number of training images, it becomes highly difficult (computationally) to calculate the optimum gradient taking all the images at once into the optimizer.

Hence, a small random sample of 128 images is selected to train the activation layer (weights and biases) in each iteration. In the current setup, we run 100 iterations and report training accuracy for every tenth iteration.

However, these can be increased based on the cluster configuration or computational power (CPU or GPU) to obtain higher model accuracy. In addition, a 50% dropout rate is used to train the CNN in each iteration. In step 6, we can evaluate the performance of the trained model on a test data of 10,000 images.

# Evaluating the performance on test data

In this recipe, we will look into the performance of the trained CNN on test images using a confusion matrix and plots.

## Getting ready

The prerequisite packages for plots are `imager` and `ggplot2`.

## How to do it...

1. Get the `actual` or `true` class labels of test images:

```
test_true_class <- c(unlist(images.lab.test))
```

2. Get the predicted class labels of test images. Remember to add `1` to each class label, as the starting index of TensorFlow (the same as Python) is 0 and that of R is `1`:

```
test_pred_class <- y_pred_cls$eval(feed_dict = dict(
x = test_data$images, y_true = test_data$labels, keep_prob = 1.0))
test_pred_class <- test_pred_class + 1
```

3. Generate the confusion matrix with rows as true labels and columns as predicted labels:

```
table(actual = test_true_class, predicted = test_pred_class)
```

4. Generate a plot of the `confusion` matrix:

```
confusion <- as.data.frame(table(actual = test_true_class,
predicted = test_pred_class))
plot <- ggplot(confusion)
plot + geom_tile(aes(x=actual, y=predicted, fill=Freq)) +
scale_x_discrete(name="Actual Class") +
scale_y_discrete(name="Predicted Class") +
scale_fill_gradient(breaks=seq(from=-.5, to=4, by=.2)) +
labs(fill="Normalized\nFrequency")
```

5. Run a helper function to plot images:

```
check.image <- function(images.rgb,index,true_lab, pred_lab) {
require(imager)
# Testing the parsing: Convert each color layer into a matrix,
# combine into an rgb object, and display as a plot
img <- images.rgb[[index]]
img.r.mat <- as.cimg(matrix(img$r, ncol=32, byrow = FALSE))
img.g.mat <- as.cimg(matrix(img$g, ncol=32, byrow = FALSE))
img.b.mat <- as.cimg(matrix(img$b, ncol=32, byrow = FALSE))
img.col.mat <- imappend(list(img.r.mat,img.g.mat,img.b.mat),"c")
# Plot with actual and predicted label
plot(img.col.mat,main=paste0("True: ", true_lab,":: Pred: ",
pred_lab),xaxt="n")
axis(side=1, xaxp=c(10, 50, 4), las=1)
}
```

6. Plot random misclassified test images:

```
labels <-
c("airplane","automobile","bird","cat","deer","dog","frog","horse",
"ship","truck")
# Plot misclassified test images
plot.misclass.images <- function(images.rgb, y_actual,
y_predicted,labels){
# Get indices of misclassified
indices <- which(!(y_actual == y_predicted))
id <- sample(indices,1)
# plot the image with true and predicted class
true_lab <- labels[y_actual[id]]
pred_lab <- labels[y_predicted[id]]
check.image(images.rgb,index=id,
true_lab=true_lab,pred_lab=pred_lab)
}
plot.misclass.images(images.rgb=images.rgb.test,y_actual=test_true_
class,y_predicted=test_pred_class,labels=labels)
```

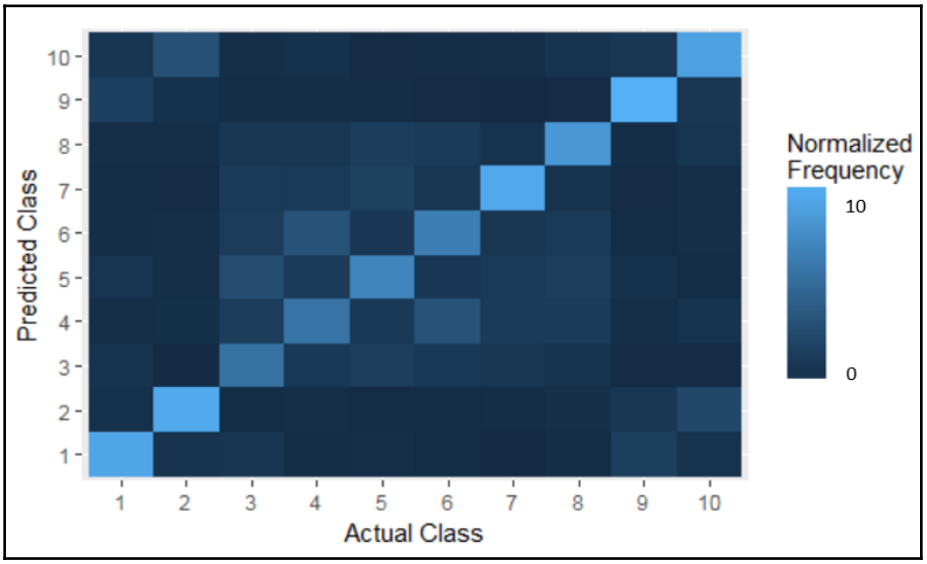
# How it works...

In steps 1 through 3, we extract the true and predicted test class labels and create a confusion matrix. The following image shows the confusion matrix of the current test predictions:

| | predicted | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| actual | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 582 | 36 | 50 | 28 | 55 | 22 | 22 | 29 | 117 | 59 |
| 2 | 46 | 598 | 9 | 30 | 24 | 24 | 13 | 24 | 38 | 194 |
| 3 | 62 | 18 | 348 | 100 | 177 | 96 | 89 | 63 | 20 | 27 |
| 4 | 21 | 25 | 80 | 368 | 88 | 211 | 83 | 66 | 21 | 37 |
| 5 | 29 | 17 | 106 | 80 | 439 | 74 | 119 | 103 | 19 | 14 |
| 6 | 17 | 17 | 80 | 211 | 70 | 416 | 65 | 89 | 12 | 23 |
| 7 | 5 | 23 | 66 | 87 | 89 | 64 | 594 | 41 | 6 | 25 |
| 8 | 23 | 32 | 49 | 89 | 106 | 82 | 41 | 520 | 11 | 47 |
| 9 | 108 | 67 | 14 | 29 | 37 | 17 | 11 | 19 | 632 | 66 |
| 10 | 41 | 151 | 12 | 48 | 16 | 27 | 24 | 56 | 63 | 562 |

The test accuracy post 700 training iterations is only ~51% and can be further improved by increasing the number of iterations, increasing the batch size, configuring layer parameters such as the number of convolution layers (used 2), types of activation functions (used ReLU), number of fully connected layers (used two), optimization objective function (used accuracy), pooling (used max 2 x 2), dropout probability, and many others.

Step 4 is used to build a facet plot of the test confusion matrix, as shown in the following screenshot:




In step 5, we define a helper function to plot the image along with a header containing both true and predicted classes. The input parameters of the `check.image` function are (`test`) flattened input dataset (`images.rgb`), image number (`index`), true label (`true_lab`), and predicted label (`pred_lab`). Here, the red, green, and blue pixels are initially parsed out, converted into a matrix, appended as a list, and displayed as an image using the *plot* function.

In step 6, we plot misclassified test images using the helper function of step 5. The input parameters of the `plot.misclass.images` function are (`test`) flattened input dataset (`images.rgb`), a vector of true labels (`y_actual`), a vector of predicted labels (`y_predicted`), and a vector of unique ordered character labels (`labels`). Here, the indices of the misclassified images are obtained and an index is randomly selected to generate the plot. The following screenshot shows a set of six misclassified images with true and predicted labels: