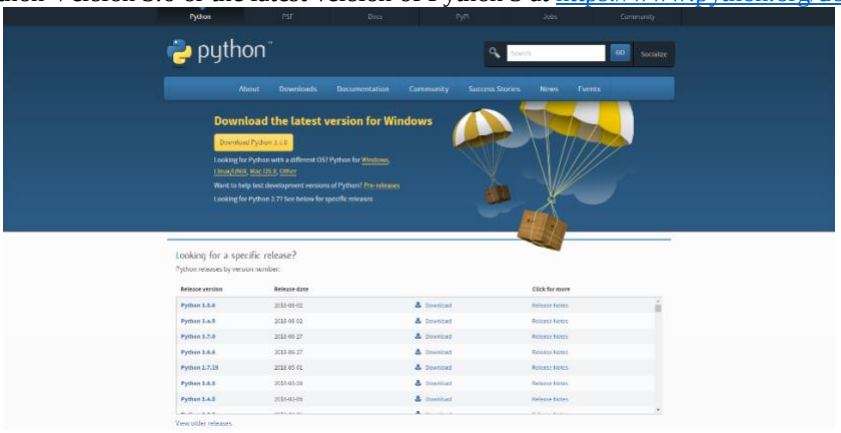
Convolutional Neural Networks Using Keras and TensorFlow: R Code
Belami E-Commerce VAM Department 2018
By: Carlos Monsivais
August 30, 2018

Table of Contents

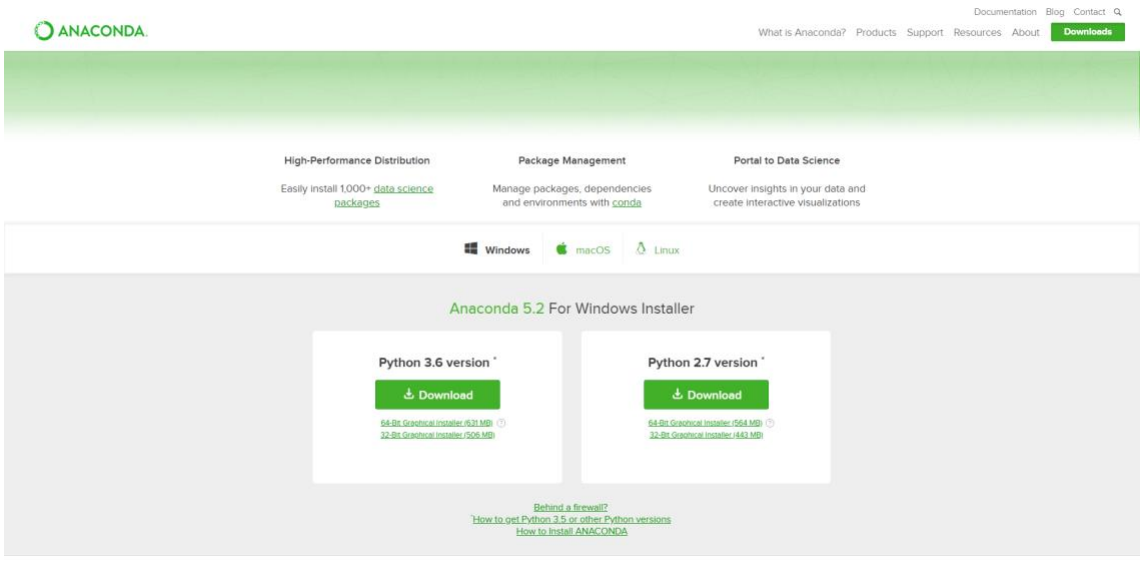
Step 1: Install Keras

To install Keras follow these steps:

1. Install Python Version 3.6 or the latest version of Python 3 at https://www.python.org/downloads/



2. Go to the following link https://www.anaconda.com/download/ and choose between downloading on Windows, macOS, or Linux. Make sure you download the 64-Bit version for the Python 3.6 version.



3.
```
install.packages("devtools")
library(devtools)
devtools::install_github("rstudio/tensorflow")
devtools::install_github("rstudio/reticulate")
library(tensorflow)
install_tensorflow()
install.packages("keras")
```

If there are any problem go to this youtube video for more detailed instructions https://www.youtube.com/watch?v=z0qhKP2liHs and to this page https://keras.rstudio.com/

Step 2: Loading the Packages

```
library(keras)
```

We need this library in order to actually create the convolutional neural networks and to use the one hot encoding method for the data we will use.

```
library(EBImage)
```

We need this library to read in the images, look at their structures such as dimensions and to resize the images.

Step 3: Reading in the Images

```
setwd("/Users/carlos.monsivais/Desktop")
```

Need to set the working directory to the place where the images are stores. Therefore, make sure to write the correct path name to the area where you have your images.

```
pic1 = c("islandlight1.jpeg", "islandlight2.jpeg", "islandlight3.jpeg", "islandlight4.jpeg", "islan
dlight5.jpeg", "islandlight6.jpeg", "islandlight7.jpeg", "islandlight8.jpeg","tracklight1.jpeg", "t
racklight2.jpeg", "tracklight3.jpeg", "tracklight4.jpeg", "tracklight5.jpeg", "tracklight6.jpeg", "
tracklight7.jpeg", "tracklight8.jpeg", "minipendant1.jpeg", "minipendant2.jpeg", "minipendant3.jpeg
", "minipendant4.jpeg", "minipendant5.jpeg", "minipendant6.jpeg", "minipendant7.jpeg", "minipendant
8.jpeg")
```

Above is the vector where the names of the images are listed. Make sure that you only list the training images you will use to train the algorithm. As a result, the images in this vector are the names of the training images only.

```
train = list()
```

Here we are creating an open vector where we will store the training images that we listed above. We are creating this open vector to read in and store the images here in the variable called train.

```
for(i in 1:24) {train[[i]] = readImage(pic1[i])}
```

Since we uploaded 24 training images into the pic1 vector we need to execute the for function from the 1rst image to the 24th image. We will be using the readImage function to read in the images that we listed in the pic1 variable however we will then store the images that were read in, into the open list we have created called train because the images we read in are considered the training images.

```
pic2 = c("islandlight9.jpeg", "islandlight10.jpeg","tracklight9.jpeg", "tracklight10.jpeg
","minipendant9.jpeg", "minipendant10.jpeg")
```

Now we want to list the names of the testing images we will be using.

```
test = list()
```

Here is the open list where we will store the testing images next.

```
for(i in 1:6) {test[[i]] = readImage(pic2[i])}
```

Since there are 6 images in the testing vector, we need to execute the function from the 1rst to the 6th image. Here we are again doing the same process we used in the training image set however we are reading the images into the test list we created to store them there.

Step 4: Analyzing the Image Components

```
print(train[[2]])
Image
  colorMode    : Color
  storage.mode : double
  dim          : 300 300 3
  frames.total : 3
  frames.render: 1

imageData(object)[1:5,1:6,1]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    1    1    1    1
[2,]    1    1    1    1    1    1
[3,]    1    1    1    1    1    1
[4,]    1    1    1    1    1    1
[5,]    1    1    1    1    1    1
```

What the plot function does is that it plots the image in the form of a matrix however the number in the [[]] is the order of which image you want to print out in the training image set. We get other information such as the dimensions, method of storage, and whether the image is in color.

```
summary(train[[2]])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  1.0000  1.0000  0.9584  1.0000  1.0000
```

The summary function on the specific image you choose to see a summary of shows the numerical general statistics regarding the images matrix values.

```
display(train[[2]])
```

Here we can display the image we choose using the numerical value regarding the image we pick however it display the image in the viewer format which means its displaying the image on your computer as the local host.

```
plot(train[[2]])
```

This will plot the image in R in the Plots section of R Studio so now it plots the image on R Studio as the local host.

```
par(mfrow = c(3,8))
```

We are using this to plot the following images in the format where it will do so in 3 rows and 8 columns.

```
for(i in 1:24)plot(train[[i]])
```

Here we can see that we plotted the 24 training images in the format that we specified above being a 3 by 8 matrix of the images.

```
par(mfrow = c(1,1))
```

We are running this to tell R that the next time we plot something we want it done in a 1 by 1 matrix fashion therefore we are plotting things one at a time.

Step 5: Scaling the Images

```
str(train)
List of 24
 $ :Formal class 'Image' [package "EBImage"] with 2 slots
  .. ..@ .Data    : num [1:300, 1:300, 1:3] 1 1 1 1 1 1 1 1 1 ...
  .. ..@ colormode: int 2
 $ :Formal class 'Image' [package "EBImage"] with 2 slots
  .. ..@ .Data    : num [1:300, 1:300, 1:3] 1 1 1 1 1 1 1 1 1 ...
  .. ..@ colormode: int 2
```

Here we can see the structure of each of the 24 images in terms of their dimensions. This is not the full list because the ful l list of all 24 images in the training image set was too long.

```
for(i in 1:24){train[[i]] = resize(train[[i]], 100, 100)}
```

Here we are resizing all 24 of the training images. We are resizing them into a format so that they all have the 100 by 100 dimensions so tat we can compare them equally when processing them through the neural networks.

```
str(train)
List of 24
 $ :Formal class 'Image' [package "EBImage"] with 2 slots
  .. ..@ .Data    : num [1:100, 1:100, 1:3] 1 1 1 1 1 1 1 1 1 ...
  .. ..@ colormode: int 2
 $ :Formal class 'Image' [package "EBImage"] with 2 slots
  .. ..@ .Data    : num [1:100, 1:100, 1:3] 1 1 1 1 1 1 1 1 1 ...
  .. ..@ colormode: int 2
```

Again, I only listed the first two parts of this command because having all the structures for the 24 images would be too much however we can see that the images were resized to fit the 100 by 100 by 3 dimensions. Now we have all the images being the exact same size.

```
for(i in 1:6){test[[i]] = resize(test[[i]], 100, 100)}
```

Now we are resizing all the images in the testing set with a size of 100 by 100 dimensions.

```
train = combine(train)
train
Image
  colorMode    : Color
  storage.mode : double
  dim          : 100 100 3 24
  frames.total : 72
  frames.render: 24

imageData(object)[1:5,1:6,1,1]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    1    1    1    1
[2,]    1    1    1    1    1    1
[3,]    1    1    1    1    1    1
[4,]    1    1    1    1    1    1
[5,]    1    1    1    1    1    1
```

We are combing al the training images into one big image. As you can see the dimensions of the one big image is the size of the scale that we used being 100 by 100 by 3 by 24 where the 24 refers to the number of images that were combined to make this big image.

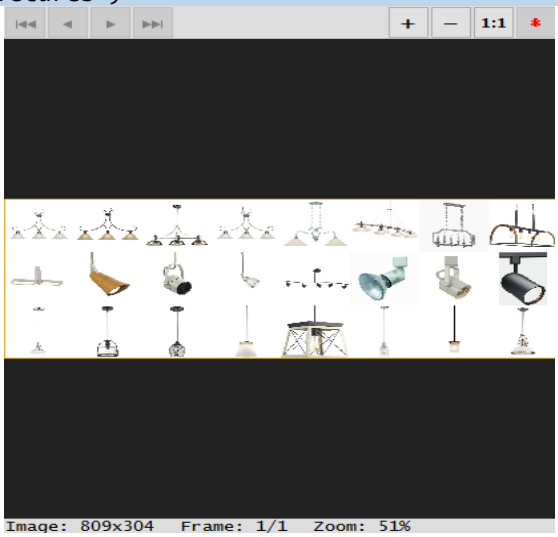
```
x = tile(train, 8)
x
```

```
Image
  colorMode    : Color
  storage.mode : double
  dim          : 809 304 3
  frames.total : 3
  frames.render: 1

imageData(object)[1:5,1:6,1]
          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 0.8941176 0.8941176 0.8941176 0.8941176 0.8941176 0.8941176
[2,] 0.8941176 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[3,] 0.8941176 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[4,] 0.8941176 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[5,] 0.8941176 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
```

By using the tile function on the training image set what we are telling it to plot the images in the forms of tiles where we will plot them in 8 images per row. When we look at the tile function x we can see how the matrix dimensions have changed now that the image is looked at as one whole image made up of these tiles. One thing to note is that you want to categorize the number of images per row. For example, in this training data set I have 8 images of island lights, 8 images of track lights, and 8 images of mini pendants therefore, I want there to be 8 images per rows so that they are already categorized.

```
display(x, title = "Training Pictures")
```



By using the display function, we are looking at how we have transformed the 24 images into one whole image in the form of 24 tiles making up the whole image we will analyze.

```
test = combine(test)
test
```

```
Image
  colorMode    : Color
  storage.mode : double
  dim          : 100 100 3 6
  frames.total : 18
  frames.render: 6

imageData(object)[1:5,1:6,1,1]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    1    1    1    1
[2,]    1    1    1    1    1    1
[3,]    1    1    1    1    1    1
[4,]    1    1    1    1    1    1
[5,]    1    1    1    1    1    1
```

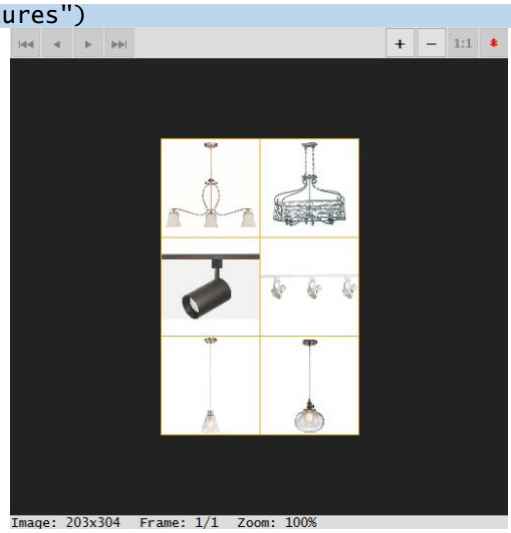
We are combining the matrixes of the 6 testing images into one big image matrix.

```
y = tile(test, 2)
y
Image
  colorMode    : Color
  storage.mode : double
  dim          : 203 304 3
  frames.total : 3
  frames.render: 1

imageData(object)[1:5,1:6,1]
          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 0.8941176 0.8941176 0.8941176 0.8941176 0.8941176 0.8941176
[2,] 0.8941176 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[3,] 0.8941176 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[4,] 0.8941176 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
[5,] 0.8941176 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
```

Here we will plot 2 images per row where if we look at the tile function y we can see that the image is represented by one whole matrix. Again I want to note that in the training data set I have 2 images of island lights, 2 images of track lights, and 2 images of mini pendants therefore I want to group these images in pairs of two so they are still separated by category.

```
display(y, title = "Testing Pictures")
```



Now we can see what the testing images look like as one whole image.

```
str(train)
Formal class 'Image' [package "EBImage"] with 2 slots
  ..@ .Data    : num [1:100, 1:100, 1:3, 1:24] 1 1 1 1 1 1 1 1 1 1 ...
  ..@ colormode: int 2
```

We can now see that since we transformed the 24 training images into one whole image, the structure of the image is still 100 by 100 by 3 (where the 3 represents it has the three main colorways such as red, green, and blue) by 24 where the 24 represents the 24 combined images.

```
str(test)
Formal class 'Image' [package "EBImage"] with 2 slots
  ..@ .Data    : num [1:100, 1:100, 1:3, 1:6] 1 1 1 1 1 1 1 1 1 1 ...
  ..@ colormode: int 2
```

Now we can see the new structure of the testing image data set where we transformed the 6 images into one whole image with the size of 100 by 100 by 3 by 6 where the 6 represents the 6 images that make up the whole image.

Step 6: Reorganizing the Dimensions

```
dim(train)
[1] 100 100   3  24
```

Here we can see the dimensions of the training data where they are 100 by 100 by 3 by 24 however we are going to need to reformat the dimensions in the format of 24 by 100 by 100 by 3 for the convolutional layers to work. We want the format to be number of images by scale by sale by colorways.

```
train = aperm(train, c(4, 1, 2, 3))
```

Here we are using the aperm function to transpose the array by permuting its dimensions. For example, in this case we want to transpose the array so that the dimensions are in the order of 24, 100, 100, 3. Therefore we want to move the 4th column in the original dimensions to the 1rst column, move the 1rst column in the original dimensions to the 2nd column, move the 2nd column of the original dimensions to the 3rd column and move the 3rd column in the original dimensions to the 4th column.

```
str(train)
num [1:24, 1:100, 1:100, 1:3] 1 1 1 1 1 ...
```

Now we can see that by using the aperm function we have reorganized the dimensions in the format that we wanted.

```
dim(test)
[1] 100 100   3   6
```

Here we can see the dimensions of the testing data are 100 by 100 by 3 by 6 however we are going to need to reformat the dimensions in the format of 6 by 100 by 100 by 3 for the convolutional layers to work.

```
test = aperm(test, c(4,1,2,3))
```

Again we are using the aperm function to transpose the array by permuting its dimensions in the same order as above.

```
str(test)
num [1:6, 1:100, 1:100, 1:3] 1 1 1 1 1 1 1 1 1 1 ...
```

Now we can see the aperm function has reorganized the dimensions in the format that we wanted.

Step 7: Labeling the Images

```
trainy = c(0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2)
```

Here we are labeling each category for the training images. For example, in the images island lights are represented by the integer 0, track lights are represented by the value 1 and mini pendants are represented by the value 2. We are assigning numerical value to the images, so we can keep track of what the algorithm predicts as.

```
testy = c(0,0,1,1,2,2)
```

Here we are labeling each category for the testing images. For example, in the images island lights are represented by the integer 0, track lights are represented by the value 1 and mini pendants are represented by the value 2.

Step 8: One Hot Encoding Image Categories

We need to use one hot encoding because the algorithm we will create is not going to interpret the image categories as island lights, or track lights, or mini pendants but will interpret them as integers such as 0, 1 or 2 which is something we have already assigned to the images. However, by putting them in the one hot encoding format, we are putting the numerical values in the form of a matrix which is made up of 1's and 0's. For example, a 1 is assigned if it does correspond to the category while a 0 is assigned if it does not correspond to that category. This is because many machine learning techniques can't operate on categorical variables and need to have numerical variables as input.

```
trainLabels = to_categorical(trainy)
trainLabels
      [,1] [,2] [,3]
 [1,]    1    0    0
 [2,]    1    0    0
 [3,]    1    0    0
 [4,]    1    0    0
 [5,]    1    0    0
 [6,]    1    0    0
 [7,]    1    0    0
 [8,]    1    0    0
 [9,]    0    1    0
[10,]    0    1    0
[11,]    0    1    0
[12,]    0    1    0
[13,]    0    1    0
[14,]    0    1    0
[15,]    0    1    0
[16,]    0    1    0
[17,]    0    0    1
[18,]    0    0    1
[19,]    0    0    1
[20,]    0    0    1
[21,]    0    0    1
[22,]    0    0    1
[23,]    0    0    1
[24,]    0    0    1
```

Here we are labeling the training data using the one hot encoding method. For example, we can see that the rows correspond to the images. The first 8 rows correspond to the island light images, the rows 9 to 16 correspond to the track light images, and the rows 17 to 24 correspond to the mini pendant image categories. As you can see the way that one hot encoding works is that it only assigns 1's and 0's where a 1 means that it does correspond to the given image category and the 0 means it does not correspond to the given image category.

```
testLabels = to_categorical(testy)
testLabels
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    1    0    0
[3,]    0    1    0
[4,]    0    1    0
[5,]    0    0    1
[6,]    0    0    1
```

Here we are labeling the testing data using the one hot encoding method. Same method as above however just on the testing data.

Step 9: Creating the Model

```
model = keras_model_sequential()
```

We are creating a space for the model we will create below. Here we will construct the sequential model by passing through a list of layered constructers such as convolutions.

```
model %>%
+
```

Here we are beginning to create the model. The %>% sign just means that we will continue writing the model on the next line however it still belongs to the variable model.

```
layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu', input_shape = c(100,100,3))
%>%
+
```

Here, we are using the function layer_conv_2d because it uses spatial convolution which means that we are going to use the convolutional process where we move the filter window across a 2-D matrix that we have in our image matrix. The following is what the variables mean:

- filters: the number of output filters in the convolution
- kernel_size: this refers to the width by the height of the filter mask
- activation: the function used to determine the output of a neural network, we are using relu because it gives us fewer neurons making the network lighter an therefore easier to use.
- input_shape: The dimensions if the matrix, in this case we made the dimensions 100 by 100 by 3.

```
layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = 'relu') %>%
+
```

We are using another 2-D convolutional layer for the model, with he same inputs as above except without the input_shape function in the model because we already defined the input_shape.

```
layer_max_pooling_2d(pool_size = c(2,2)) %>%
+
```

We apply a pooling layer so that we can reduce the complexity and the dimensions of the model. The following functions is :

- pool_size : the size of the pooled matrix

```
layer_dropout(rate = 0.25) %>%
+
```

We are using the layer dropout for regularization so there is no overfitting. By doing so we are trading training performance for more generalization. It works in how during the training half of the neurons will be deactivated which improves generalization for predictions because it forces your neurons to learn the same concept with different neurons.

- rate : a value between 0 and 1 telling the model the fraction of the input to drop to not overfit.

```
layer_conv_2d(filters  =64, kernel_size = c(3,3), activation = 'relu') %>%
+
```

We are applying another convolution layer, it has the same features as above except this time we have increased the filters for this layer.

```
layer_conv_2d(filters  =64, kernel_size = c(3,3), activation = 'relu') %>%
+
```

We are applying another convolution layer, it has the same features as above.

```
layer_max_pooling_2d(pool_size = c(2,2)) %>%
+
```

We are again applying a pooling layer the same as the one above to reduce the complexity of the dimensions.

```
layer_dropout(rate = 0.25) %>%
+
```

Again, we are applying the dropout layer to not overfit, same dropout layer as above.

```
layer_flatten() %>%
+
```

We are using this function we are about to create to flatten the output from the convolution layers. For example, we want to make them into a 1-D and therefore convert the neurons into a vector.

```
layer_dense(units = 256, activation = 'relu') %>%
+
```

This is where we tell the model how many neurons to create when it is flattened.

- units: The dimensionality of the output space, how many neurons do you want your output space to have.
- activation: The name of the activation function to use, usually should use relu.

```
layer_dropout(rate = 0.25) %>%
+
```

Like before we need to use the layer_dropout function to not overfit, it has the same inputs as before.

```
layer_dense(units = 3, activation = 'softmax')%>%
+
```

This part is very important. We are again using the layer_dense function however this time we have 3 units because we have 3 different image categories. Therefore, the number of units must be the same as the number of categories in this specific part of the model. We are also using the softmax activation function because it is often used in the final layer of a neural network classifier because this function gives us the ability to see the results in the form of a probability.

- units: in this final layer it must be the number of categories you have.
- activation: in this final layer it should be the softmax function because it lets you see the results in the form of a probability which is something that is easily understood. For example, lets you see the probability of a categorization.

```
compile(loss = 'categorical_crossentropy', optimizer = optimizer_sgd(lr = 0.01, decay = 1e-6,
momentum = 0.9, nesterov =TRUE), metrics = c('accuracy'))
```

Now we are compiling our model to update its parameters. Therefore we are configuring the model for training in the next phase. The model consists of:

- loss: This is the loss function used to measure the inconsistency between the predicted value and the actual value. We are using the categoricl_crossentropy because we want to classify many categorical variables, in this case images.
- optimizer: the optimization instance, here we are using the optimizer_sgd because it performs Stochastic Gradient Descent which is where we use the whole training dataset to perform the gradient descent. Since we have a small image set we can use this but with massive image sets this would be very costly.
    - lr: this is the learning rate, generally you optimize your model with a large learning rate of 0.1, and then progressively reduce this rate, often by an order of magnitude of 0.01, then 0.001 and so on.
    - decay: the decay of each learning rate every time it is updated.
    - momentum: the parameter that accelerate the optimizer_sgd when it is optimizing in the right direction and makes back and forth movements a lot weaker when trying to optimize.
    - nesterov: Whether we should or should not apply nesterov movement. Nesterov movement is when it predicts whether momentum will continue or not and adjust for it. Therefore, we want this on.
- metrics : this is the list of the metrics to be evaluated by the model during the training and testing phase. We want to use accuracy because we want to see how the model does.

```
summary(model)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_69 (Conv2D)           (None, 98, 98, 32)        896
_________________________________________________________________
conv2d_70 (Conv2D)           (None, 96, 96, 32)        9248
_________________________________________________________________
max_pooling2d_35 (MaxPooling2D) (None, 48, 48, 32)     0
_________________________________________________________________
dropout_52 (Dropout)         (None, 48, 48, 32)        0
_________________________________________________________________
conv2d_71 (Conv2D)           (None, 46, 46, 64)        18496
_________________________________________________________________
conv2d_72 (Conv2D)           (None, 44, 44, 64)        36928
_________________________________________________________________
max_pooling2d_36 (MaxPooling2D) (None, 22, 22, 64)     0
_________________________________________________________________
dropout_53 (Dropout)         (None, 22, 22, 64)        0
_________________________________________________________________
flatten_18 (Flatten)         (None, 30976)             0
_________________________________________________________________
dense_35 (Dense)             (None, 256)               7930112
_________________________________________________________________
dropout_54 (Dropout)         (None, 256)               0
_________________________________________________________________
dense_36 (Dense)             (None, 3)                 771
=================================================================
Total params: 7,996,451
Trainable params: 7,996,451
Non-trainable params: 0
_________________________________________________________________
```

We can see the summary of the model we just created. The interesting attributes is to see in the shape column how we reduce the dimensionality of the complex training data matrix into a 1-Dimensional matrix. Also, to see how we start out looking at more and more parameters until we reduce them again.

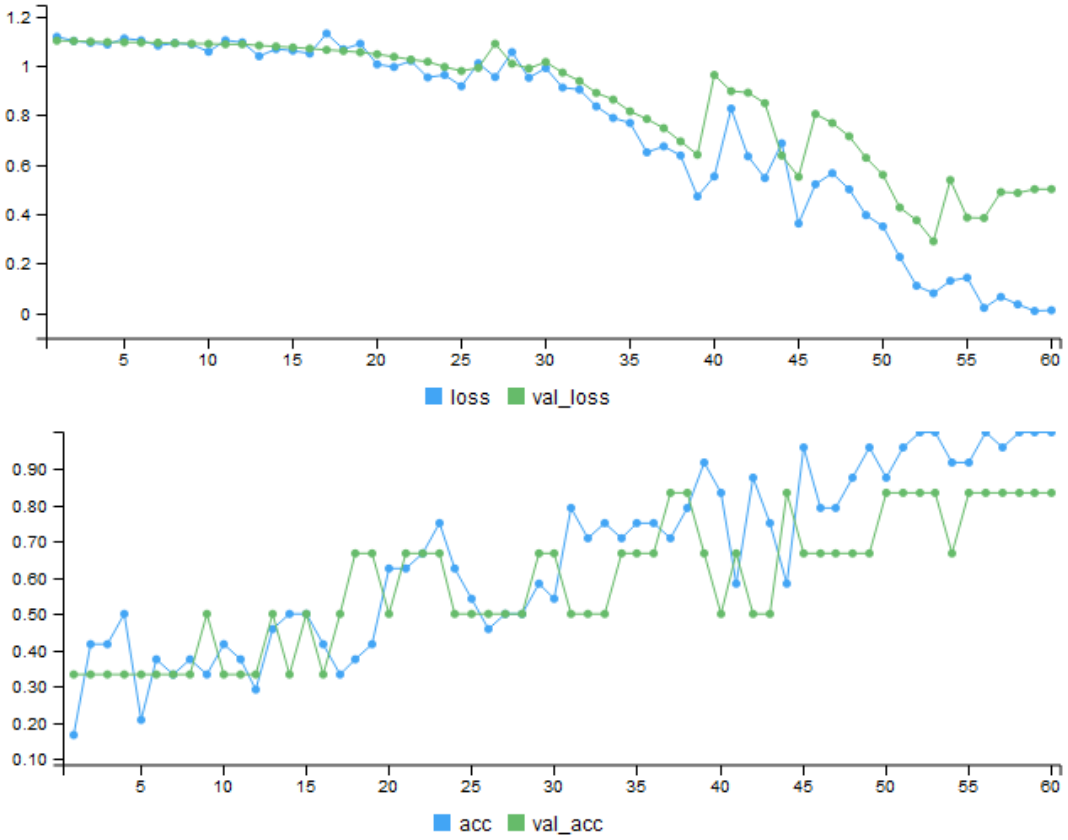
Step 10: Fitting the Model

Now we can fit the model in the sense of using the training images in order train the convoluted model we just created.

```
history = model %>%
+
```

Now that we are training our model with the training images we are calling this fitted model history.

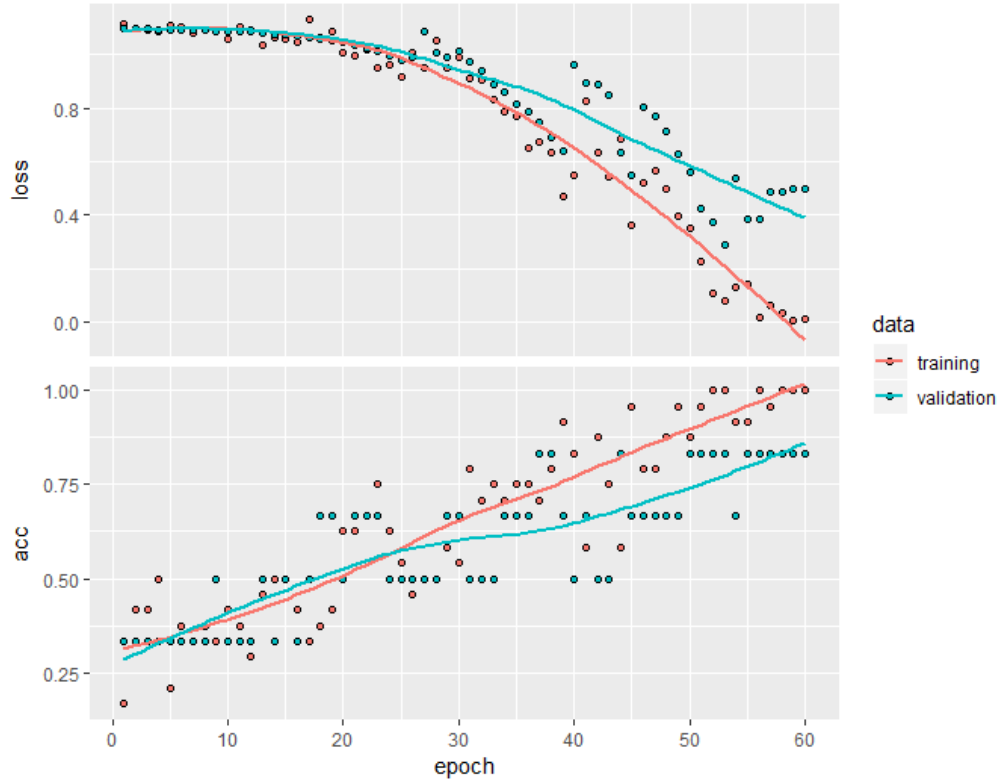
```
fit(x = train, y = trainLabels, epochs = 60, batch_size = 32, validation_data = list(test,
testLabels))
Train on 24 samples, validate on 6 samples
Epoch 1/60
24/24 [==============================] - 5s 227ms/step - loss: 1.1170 - acc: 0.1667 - val_loss:
1.1012 - val_acc: 0.3333
Epoch 2/60
24/24 [==============================] - 1s 52ms/step - loss: 1.1007 - acc: 0.4167 - val_loss:
1.0998 - val_acc: 0.3333
```



We are now fitting the model to the training data and validating it to see how well it will perform against the testing images. The input for the function is:

- x: The array of training image matrixes.
- y: The training image one hot encoder matrix.
- epochs: The number of iterations over the entire x and y data provided. This is like an iterative algorithm during the learning phase when it is getting fit for the data.
- batch_size: The number of samples for each gradient update. The default is 32.
- validation_data: The list where you want to first put the array of testing image matrixes and then the testing image one hot encoder. This will tell evaluate the loss function at the end of each epoch.

```
plot(history)
```



This plots the accuracy and loss functions per epoch to see how well your model will perform. We are plotting the model that has already been fitted to the training images.

Step 11: Evaluation on the Training Data

Just to see how well the model fits the images that we fed into it we will evaluate it based on the training images, so the images we sued to create the model we want to see how well the model fit.

```
model %>% evaluate(train, trainLabels)
24/24 [==============================] - 0s 18ms/step
$`loss`
[1] 0.003377957

$acc
[1] 1
```

This is the evaluation the including the los value and accuracy metric for the model when we are testing it. We are using the training images just to see how well the model performed.

```
predtrain = model %>% predict_classes(train)
predtrain
 [1] 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2
```

We are using the predict_classes function to generate class predictions for the training image samples. Note that we are using this function on the matrix array for the training images, not on the one hot encoded image matrix.

```
table(Predicted = predtrain, Actual = trainy)
         Actual
Predicted 0 1 2
        0 8 0 0
        1 0 8 0
        2 0 0 8
```

Here we are creating a confusion matrix to see how well our model performed. Where the following variables in the function are:

- Predicted: The predicted values we got as output based on the predicted function we created.
- Actual: The actual values that correspond to the predicted values

```
prob = model %>%
+     predict_proba(train)
```

Generates class probability predictions for the input training image samples.

```
cbind(round(prob,4), Predicted_class = predtrain, Actual = trainy)
                          Predicted_class Actual
 [1,] 0.9974 0.0025 0.0001               0      0
 [2,] 1.0000 0.0000 0.0000               0      0
 [3,] 0.9997 0.0000 0.0003               0      0
 [4,] 0.9974 0.0025 0.0001               0      0
 [5,] 0.9936 0.0020 0.0044               0      0
 [6,] 0.9889 0.0111 0.0000               0      0
 [7,] 1.0000 0.0000 0.0000               0      0
 [8,] 1.0000 0.0000 0.0000               0      0
 [9,] 0.0178 0.9736 0.0086               1      1
[10,] 0.0001 0.9998 0.0001               1      1
[11,] 0.0000 1.0000 0.0000               1      1
[12,] 0.0005 0.9994 0.0000               1      1
[13,] 0.0000 1.0000 0.0000               1      1
[14,] 0.0001 0.9983 0.0016               1      1
[15,] 0.0008 0.9991 0.0001               1      1
[16,] 0.0000 1.0000 0.0000               1      1
[17,] 0.0004 0.0001 0.9996               2      2
[18,] 0.0000 0.0000 1.0000               2      2
[19,] 0.0000 0.0000 1.0000               2      2
[20,] 0.0005 0.0003 0.9991               2      2
[21,] 0.0000 0.0000 1.0000               2      2
[22,] 0.0092 0.0171 0.9737               2      2
[23,] 0.0000 0.0000 1.0000               2      2
[24,] 0.0000 0.0000 1.0000               2      2
```

We are combining the probabilities of the predictions being assigned to the class the model is predicting and comparing the predicted class compared to the actual class for the training images.

Step 12: Evaluation on the Testing Data

Now we are going to evaluate the model based on the testing images which it has never seen. It only used them to calculate the loss function but did not include them when creating or fitting the model. The same commands from above apply however just to the testing data such as the testLabels, testy, and test variables instead of the same labels corresponding to the training data.

```
model %>% evaluate(test, testLabels)
6/6 [==============================] - 0s 18ms/step
$`loss`
[1] 0.5001166

$acc
[1] 0.8333333
```

This is the evaluation the including the los value and accuracy metric for the model when we are testing it. We are using the testing images just to see how well the model performed.

```
predtest = model %>% predict_classes(test)
predtest
[1] 2 0 1 1 2 2
```

We are using the predict_classes function to generate class predictions for the testing image samples. Note that we are using this function on the matrix array for the testing images, not on the one hot encoded image matrix.

```
table(Predicted = predtest, Actual = testy)
         Actual
Predicted 0 1 2
        0 1 0 0
        1 0 2 0
        2 1 0 2
```

Here we are creating a confusion matrix to see how well our model performed. Where the following variables in the function are:

- Predicted: The predicted values we got as output based on the predicted function we created.
- Actual: The actual values that correspond to the predicted values

```
prob = model %>%
+     predict_proba(test)
```

Generates class probability predictions for the input testing image samples.

```
cbind(round(prob,4), Predicted_class = predtest, Actual = testy)
                     Predicted_class Actual
[1,] 0.0562 0.0015 0.9423               2      0
[2,] 0.9990 0.0009 0.0000               0      0
[3,] 0.0151 0.9845 0.0004               1      1
[4,] 0.0417 0.9476 0.0106               1      1
[5,] 0.0114 0.0392 0.9494               2      2
[6,] 0.0000 0.0000 1.0000               2      2
```

We are combining the probabilities of the predictions being assigned to the class the model is predicting and comparing the predicted class compared to the actual class for the testing images.

test

Conclusion

In conclusion, we can train a model to categorize images with a relatively high accuracy. We use a convolutional neural network which consist of using the process of using a convolution layer, then pooling, then applying dropout, flattening down the values to 1 dimension, and then compiling the model. We use this process by using the TensorFlow and Keras tools that can be used in R Studio where Google and JJ Allaire (the creator of R Studio) contributed along with many others to provide these powerful tools. This tool can be used not only for image recognition but can be adjusted for using the same convolution techniques to create a neural network in order to make a predictive neural network algorithm. As a result, this tool is vey powerful when it comes down to predictive power as it uses very advanced algorithms to do this.