Neural networks have really taken over for solving image recognition and high sample rate data problems in the last couple of years. In all honesty, I promise I won’t be teaching you what neural networks are or CNN’s are. There are hundred’s of resources that are published everyday explaining them. I’ll post few links below.

I am a serious R user and very new to Deep learning domain. As I started coming across new image classification projects, I started to incline towards CNN’s. I went over few tutorials regarding image classification using CNN’s and read a few books. After a few, I started to see the same old pattern in every blog post

1. Download data set
2. Split them three ways (train/test/validation)
3. Create a model (in most cases pre-trained models)
4. Set up generators
5. Compile the model
6. Predict
7. End

I understood the concept of filter, filter size and activation functions. But, I was curious on what the network was actually seeing through the filter. I did a lot of digging and found a [stackoverflow](https://datascience.stackexchange.com/questions/18751/r-keras-how-to-get-output-of-a-hidden-layer) post linking to [RStudio’s Keras-FAQ](https://keras.rstudio.com/articles/faq.html). It was literally 3 lines of code to visualize what was happening at each layer. Meanwhile in python it was over two dozen lines of code. (Irony!) I thought there might be quite a few people out there who would be interesting in knowing this in R just like me. So, I decided to write this blog post. It would be very useful when you are explaining this to your boss or a work colleague.

Let’s get started!

**Initial Setup**

**Downloading Data Set**

For this example, I will be using [cats and dogs data set from Kaggle](https://www.kaggle.com/c/dogs-vs-cats). You can follow the link and download the data. You might have to create an account to download it.

If you have your own data then don’t worry about this step. Skip it.

**Load Keras library**

library(keras)

**Split the data into train and test**

original\_dataset\_dir = "/home/rstudio/train"

base\_dir = "/home/rstudio/data"

dir.create(base\_dir)

train\_dir = file.path(base\_dir, "train")

dir.create(train\_dir)

test\_dir = file.path(base\_dir, "test")

dir.create(test\_dir)

train\_cats\_dir = file.path(train\_dir, "cats")

dir.create(train\_cats\_dir)

train\_dogs\_dir = file.path(train\_dir, "dogs")

dir.create(train\_dogs\_dir)

test\_cats\_dir = file.path(test\_dir, "cats")

dir.create(test\_cats\_dir)

test\_dogs\_dir = file.path(test\_dir, "dogs")

dir.create(test\_dogs\_dir)

fnames = paste0("cat.", 1:2000, ".jpg")

file.copy(file.path(original\_dataset\_dir, fnames),

file.path(train\_cats\_dir))

fnames = paste0("cat.", 2001:3000, ".jpg")

file.copy(file.path(original\_dataset\_dir, fnames),

file.path(test\_cats\_dir)) fnames = paste0("dog.", 1:2000, ".jpg")

file.copy(file.path(original\_dataset\_dir, fnames),

file.path(train\_dogs\_dir)) fnames = paste0("dog.", 2001:3000, ".jpg")

file.copy(file.path(original\_dataset\_dir, fnames),

file.path(test\_dogs\_dir))

**Set initial parameters**

I am creating few variables and assigning them values here. The main reason is, it’s easy to tweak them and retrain the models.

# set path

path = "/home/rstudio/data/"

# set inital parameters

img\_width = 150

img\_height = 150

channels = 3

output\_n = 2

train\_samples = length(list.files(paste0(path,"train/cats"))) + length(list.files(paste0(path,"train/dogs")))

test\_samples = length(list.files(paste0(path,"test/cats"))) + length(list.files(paste0(path,"test/dogs")))

batch\_size = 50

# set dataset directory

train\_dir = paste0(path,"train")

test\_dir = paste0(path,"test")

**Create a custom mode**

I could use a pre-trained model such as [VGG16 or VGG18](https://keras.rstudio.com/reference/application_vgg.html). But, what’s the fun in that? Let me build my own. Don’t judge me about bad layering. I am still learning.

# CNN model

model = keras\_model\_sequential() %>%

layer\_conv\_2d(filters = 8, kernel\_size = c(3,3), activation = "relu", input\_shape = c(img\_width,img\_height,channels)) %>%

layer\_conv\_2d(filters = 16, kernel\_size = c(3,3), activation = "relu") %>%

layer\_max\_pooling\_2d(pool\_size = c(2,2)) %>%

layer\_conv\_2d(filters = 32, kernel\_size = c(3,3), activation = "relu") %>%

layer\_max\_pooling\_2d(pool\_size = c(2,2)) %>%

layer\_conv\_2d(filters = 64, kernel\_size = c(3,3), activation = "relu") %>%

layer\_max\_pooling\_2d(pool\_size = c(2,2)) %>%

layer\_conv\_2d(filters = 16, kernel\_size = c(3,3), activation = "relu") %>%

layer\_flatten() %>%

layer\_dense(units = 64, activation = "relu") %>%

layer\_dense(units = 64, activation = "relu") %>%

layer\_dropout(rate = 0.3) %>%

layer\_dense(units = 128, activation = "relu") %>%

layer\_dense(units = 128, activation = "relu") %>%

layer\_dropout(rate = 0.3) %>%

layer\_dense(units = 256, activation = "relu") %>%

layer\_dense(units = 256, activation = "relu") %>%

layer\_dropout(rate = 0.3) %>%

layer\_dense(units = 64, activation = "relu") %>%

layer\_dense(units = 64, activation = "relu") %>%

layer\_dropout(rate = 0.3) %>%

layer\_dense(units = 32, activation = "relu") %>%

layer\_dense(units = output\_n, activation = "softmax")

# summary of the overall model

summary(model)

**Image processing**

**Setup image augmentation**

When your data set is small, augmentation helps in increasing your own data set. Here we have few parameters like rotation, shift and zoom that would be added to your current train set to increase your train size.

# Train data image preprocessing

datagen = image\_data\_generator(

rotation\_range = 40,

width\_shift\_range = 0.2,

height\_shift\_range = 0.2,

shear\_range = 0.2,

zoom\_range = 0.2,

horizontal\_flip = TRUE,

fill\_mode = "nearest",

samplewise\_std\_normalization = TRUE

)

**Setup image generators**

Flow from image directory really helps in easing up the pre-processing. In the previous step we put our images into separate directories based on classes. Now this function would read the images from as per each class. No need to create any metadata.

# get all the train set

train\_generator = flow\_images\_from\_directory(

train\_dir,

datagen,

color\_mode = "rgb",

target\_size = c(img\_width, img\_height),

batch\_size = batch\_size,

class\_mode = "categorical",

shuffle = TRUE

)

# Get test data set

test\_generator = flow\_images\_from\_directory(

test\_dir,

datagen,

color\_mode = "rgb",

target\_size = c(img\_width, img\_height),

batch\_size = batch\_size,

class\_mode = "categorical",

shuffle = TRUE

)

**Compile and fit the model**

Now, that we have a model and generators, we can compile the model and fit the generator. I ran the model at 100 epochs couple of times and achieved an average accuracy of 80%. Not too bad for this test model!.

# compile the model

model %>% compile(

loss = "binary\_crossentropy",

optimizer = optimizer\_adamax(lr = 0.001, decay = 0),

metrics = c("accuracy")

)

history = model %>% fit\_generator(

train\_generator,

steps\_per\_epoch = as.integer(train\_samples/batch\_size),

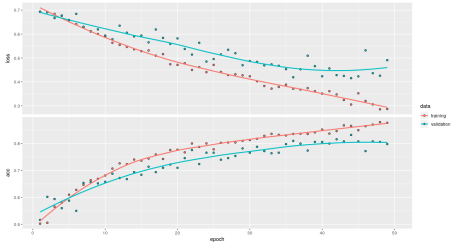
epochs = 100,

validation\_data = test\_generator,

validation\_steps = 10,

initial\_epoch = 1

)



**Visualizing a test image through layers**

Now, that we have a model, we can choose a test image and pass it through different layers that we have and visualize what happens to our image at each layer. This will tell us what was the point in the object that our network is learning from. Below we have chosen a test image that we want to test our model.

# load image

x = image\_load(paste0(path,"test/cats/cat.2001.jpg"),target\_size = c(img\_width, img\_height))

data = x %>% array\_reshape(c(-1,img\_width, img\_height, channels))

image = jpeg::readJPEG(paste0(path,"test/cats/cat.2511.jpg"))



Next, we will capture an intermediate layer, save that layer as model, predict our image based on intermediate later. We will get a multidimensional matrix output. In the below results we have an image size of 33 x 33 and 64 filters. You can tweak them to plot the results.

*Note: Index is the layer number that we want to look at.*

# what layer do we want to look at?

index = 6

# choose that layer as model

intermediate\_layer\_model = keras\_model(inputs = model$input,

outputs = get\_layer(model, index = index)$output)

# predict on that layer

intermediate\_output = predict(intermediate\_layer\_model, data)

# dimensionso of prediction

dim(intermediate\_output)

[1] 1 33 33 64

Finally, we can plot our matrix data from each of our filters into a grid using image function as shown below.

*Note: the images below are rotated. You can rotate the images using matrix rotate function.*

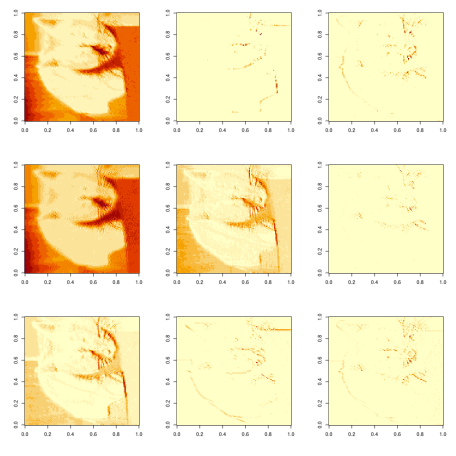
par(mfrow = c(3,3))

for(i in 1:9){

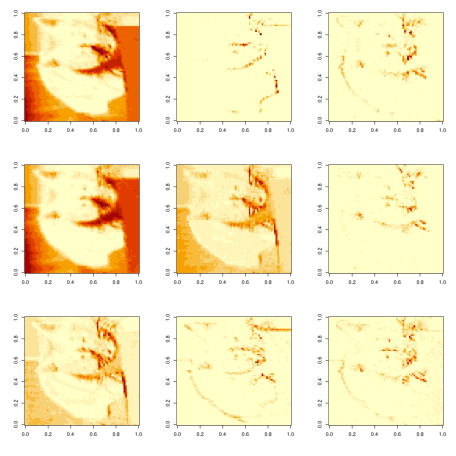
image(intermediate\_output[1,,,i])

}

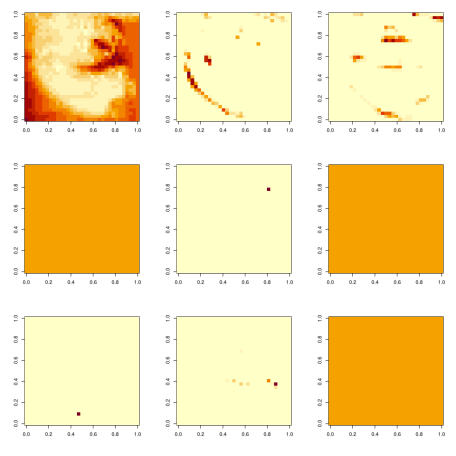
**Layer 2**



**Layer 3**



**Layer 6**



**Conclusion**

From the above you can see how the CNN filters are narrowing the point of interest to the cat. This not only helps explain how your model is working but, also a way to confirm that your model is working like it is intended to. It was quite a journey for me to go through the inner webs to find a way to visualize my layers. Hope you’all can use this for your projects.