**What is TensorFlow and Keras ?**

[**TensorFlow**](https://www.tensorflow.org/) is an open-source software library for Machine Intelligence that allows you to deploy computations to multiple CPUs or GPUs. It was developed by researchers and engineers working on the Google Brain Team.

[**Keras**](https://keras.io/) is a high-level neural networks API capable of running on top of multiple back-ends including: TensorFlow, CNTK, or Theano. One of its biggest advantages is its “user friendliness”. With Keras you can easily build advanced models like convolutional or recurrent neural network.

To install TensorFlow and Keras from R use **install\_keras()** function. If you want to use the GPU version you have to install some prerequisites first. This could be difficult but it is worth the extra effort when dealing with larger and more elaborate models. I strongly recommend you to do this! You can read more [here](https://tensorflow.rstudio.com/installation_gpu.html#prerequisites).

install.packages("keras")

library(keras)

# Make sure to install required prerequisites, before installing Keras using the commands below:

install\_keras() # CPU version

install\_keras(tensorflow = "gpu") # GPU version

**Data preparation**

For the task we will use a [dataset of 2800 satellite pictures from Kaggle](https://www.kaggle.com/rhammell/ships-in-satellite-imagery/data). Every row contains information about one photo (80-pixel height, 80-pixel width, 3 colors – RGB color space). To input data into a Keras model, we need to transform it into a 4-dimensional array (index of sample, height, width, colors). Every picture is associated with a label that could be equal **1** for a **ship** and **0** for **non-ship** object. Also here we have to use some transformations to create a binary matrix for Keras.

library(keras)

library(tidyverse)

library(jsonlite)

library(abind)

library(gridExtra)

ships\_json <- fromJSON("ships\_images/shipsnet.json")[1:2]

ships\_data <- ships\_json$data %>%

apply(., 1, function(x) {

r <- matrix(x[1:6400], 80, 80, byrow = TRUE) / 255

g <- matrix(x[6401:12800], 80, 80, byrow = TRUE) / 255

b <- matrix(x[12801:19200], 80, 80, byrow = TRUE) / 255

list(array(c(r,g,b), dim = c(80, 80, 3)))

}) %>%

do.call(c, .) %>%

abind(., along = 4) %>%

aperm(c(4, 1, 2, 3))

ships\_labels <- ships\_json$labels %>%

to\_categorical(2)

rm(ships\_json)

dim(ships\_data)

[1] 2800 80 80 3

Now we can take a look at some sample of our data. Notice that if a ship appeared partially on a picture, then it wasn’t labeled as a 1.

xy\_axis <- data.frame(x = expand.grid(1:80, 80:1)[, 1],

y = expand.grid(1:80, 80:1)[, 2])

set.seed(1111)

sample\_plots <- sample(1:dim(ships\_data)[1], 12) %>%

map(~ {

plot\_data <- cbind(xy\_axis, r = as.vector(t(ships\_data[.x, , , 1])),

g = as.vector(t(ships\_data[.x, , , 2])),

b = as.vector(t(ships\_data[.x, , , 3])))

ggplot(plot\_data, aes(x, y, fill = rgb(r, g, b))) + guides(fill = FALSE) +

scale\_fill\_identity() + theme\_void() + geom\_raster(hjust = 0, vjust = 0) +

ggtitle(ifelse(ships\_labels[.x, 2], "Ship", "Non-ship"))

})

do.call("grid.arrange", c(sample\_plots, ncol = 4, nrow = 3))

The last thing we have to do is to split our data into training and test sets.

set.seed(1234)

indexes <- sample(1:nrow(ships\_labels), 0.7 \* nrow(ships\_labels))

train <- list(data = ships\_data[indexes, , , ], labels = ships\_labels[indexes, ])

test <- list(data = ships\_data[-indexes, , , ], labels = ships\_labels[-indexes, ])

**Modeling**

In Keras you can build models in 3 different ways using:

1. a sequential model
2. functional API
3. pre-trained models

For now, we will only use sequential models. But before that, we have to understand the basic concepts behind convolutional neural networks.

**Convolutional neural networks (CNN)** or **ConvNets** are a class of deep, feed-forward artificial neural networks designed for solving problems like image/video/audio recognition, and object detection etc. The architecture of ConvNets differs depending on the issue, but there are some basic commonalities.

The first type of layer in CNN’s is a **convolutional layer** and it is a core building block of ConvNets. Simply put, we take a small set of **filters** (also called **kernels**) and place them on part of our original image to get the dot product between kernels and corresponding image parts. Next, we move our filter to the next position and repeat this action. The number of pixels that we move the filters is called a **stride**. After getting the dot product for the whole image, we get a so-called **activation map**.

The second type of layer in CNN’s is the **pooling layer**. This layer is responsible for dimensionality reduction of activation maps. There are several types of pooling, but **max pooling** is most commonly used. As it was in the case of convolutional layers, we have some filter and strides. After placing the filter on an image part, we take the maximum value from that part and move to the next region by the number of pixels, specified as strides.

The third type of layer in CNN’s is called the **activation layer**. In this layer, values from activation maps are transformed by some activation function. There are several functions to use but most common one is called a **rectified linear unit (ReLU)**.

The fourth type of layer is called a **densely (fully) connected layer** which is a classical output layer known as a feed-forward neural networks. This fully connected layer is placed at the end of a ConvNet.

We begin by creating an empty sequential model

model <- keras\_model\_sequential()

summary(model)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=======================================================================================

Total params: 0

Trainable params: 0

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Now we can add some additional layers. Note that objects in Keras are **modified in-place** so there’s no need for consecutive assignment. In the first layer, we have to specify the shape of our data.

model %>%

# 32 filters, each size 3x3 pixels

# ReLU activation after convolution

layer\_conv\_2d(

input\_shape = c(80, 80, 3),

filter = 32, kernel\_size = c(3, 3), strides = c(1, 1),

activation = "relu") %>%

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

layer\_conv\_2d(filter = 64, kernel\_size = c(3, 3), strides = c(1, 1),

activation = "relu") %>%

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

layer\_flatten() %>%

layer\_dense(2, activation = "softmax")

summary(model)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=======================================================================================

conv2d\_1 (Conv2D) (None, 78, 78, 32) 896

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2D) (None, 39, 39, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 37, 37, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2D) (None, 18, 18, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_1 (Flatten) (None, 20736) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 2) 41474

=======================================================================================

Total params: 60,866

Trainable params: 60,866

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

After building the architecture for our CNN, we have to configure it for training. We must specify the loss function, optimizer and additional metrics for evaluation. For example, we can use stochastic gradient descent as an optimization method and cross-entropy as a loss function.

model %>% compile(

loss = "categorical\_crossentropy",

optimizer = optimizer\_sgd(lr = 0.0001, decay = 1e-6),

metrics = "accuracy"

)

Finally, we are ready to fit the model but there is one more thing we can do. If we want to have a good and quick visualization of our results, we can run a visualization tool called TensorBoard.

tensorboard("logs/ships")

ships\_fit <- model %>% fit(x = train[[1]], y = train[[2]], epochs = 20, batch\_size = 32,

validation\_split = 0.2,

callbacks = callback\_tensorboard("logs/ships"))

...

Epoch 20/20

32/1567 [..............................] - ETA: 0s - loss: 0.4627 - acc: 0.7812

160/1567 [==>...........................] - ETA: 0s - loss: 0.5256 - acc: 0.7500

288/1567 [====>.........................] - ETA: 0s - loss: 0.5268 - acc: 0.7431

448/1567 [=======>......................] - ETA: 0s - loss: 0.5401 - acc: 0.7299

608/1567 [==========>...................] - ETA: 0s - loss: 0.5375 - acc: 0.7319

768/1567 [=============>................] - ETA: 0s - loss: 0.5389 - acc: 0.7305

896/1567 [================>.............] - ETA: 0s - loss: 0.5312 - acc: 0.7377

1056/1567 [===================>..........] - ETA: 0s - loss: 0.5259 - acc: 0.7453

1216/1567 [======================>.......] - ETA: 0s - loss: 0.5294 - acc: 0.7401

1376/1567 [=========================>....] - ETA: 0s - loss: 0.5217 - acc: 0.7471

1536/1567 [============================>.] - ETA: 0s - loss: 0.5191 - acc: 0.7507

1567/1567 [==============================] - 1s 484us/step - loss: 0.5188 - acc: 0.7511 - val\_loss: 0.5288 - val\_acc: 0.7449

The last thing to do is to get evaluation metrics and predictions form the test set.

predicted\_probs <- model %>%

predict\_proba(test[[1]]) %>%

cbind(test[[2]])

head(predicted\_probs)

model %>% evaluate(test[[1]], test[[2]])

set.seed(1111)

sample\_plots <- sample(1:dim(test[[1]])[1], 12) %>%

map(~ {

plot\_data <- cbind(xy\_axis, r = as.vector(t(test[[1]][.x, , , 1])),

g = as.vector(t(test[[1]][.x, , , 2])),

b = as.vector(t(test[[1]][.x, , , 3])))

ggplot(plot\_data, aes(x, y, fill = rgb(r, g, b))) + guides(fill = FALSE) +

scale\_fill\_identity() + theme\_void() + geom\_raster(hjust = 0, vjust = 0) +

ggtitle(ifelse(test[[2]][.x, 2], "Ship", "Non-ship")) +

labs(caption = paste("Ship prob:", round(predicted\_probs[.x, 2], 6))) +

theme(plot.title = element\_text(hjust = 0.5))

})

do.call("grid.arrange", c(sample\_plots, ncol = 4, nrow = 3))

[,1] [,2] [,3] [,4]

[1,] 0.04486139 0.95513862 0 1

[2,] 0.92640823 0.07359175 0 1

[3,] 0.26848912 0.73151088 0 1

[4,] 0.51208550 0.48791450 0 1

[5,] 0.15906605 0.84093398 0 1

[6,] 0.66976833 0.33023167 0 1

32/841 [>.............................] - ETA: 0s

384/841 [============>.................] - ETA: 0s

736/841 [=========================>....] - ETA: 0s

841/841 [==============================] - 0s 162us/step

$loss

[1] 0.5235391

$acc

[1] 0.7502973

As we can see, the model leaves room for improvement. It has a low accuracy (.075) and a high cross entropy loss (0.52). It is, however, a good introduction and start to Keras. We are going to explore ways of improving the network and achieving better results in part two. See you soon!