**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 below:

Local GPU

Overview

TensorFlow can be configured to run on either CPUs or GPUs. The CPU version is much easier to install and configure so is the best starting place especially when you are first learning how to use TensorFlow. Here’s the guidance on CPU vs. GPU versions from the TensorFlow website:

* TensorFlow with CPU support only. If your system does not have a NVIDIA® GPU, you must install this version. Note that this version of TensorFlow is typically much easier to install (typically, in 5 or 10 minutes), so even if you have an NVIDIA GPU, we recommend installing this version first.
* TensorFlow with GPU support. TensorFlow programs typically run significantly faster on a GPU than on a CPU. Therefore, if your system has a NVIDIA® GPU meeting the prerequisites shown below and you need to run performance-critical applications, you should ultimately install this version.

So if you are just getting started with TensorFlow you may want to stick with the CPU version to start out, then install the GPU version once your training becomes more computationally demanding.

The prerequisites for the GPU version of TensorFlow on each platform are covered below. Once you’ve met the prerequisites installing the GPU version in a single-user / desktop environment is as simple as:

[library](https://rdrr.io/r/base/library.html)(tensorflow)

[install\_tensorflow](https://tensorflow.rstudio.com/tensorflow/reference/install_tensorflow.html)(version = "gpu")

If you are using Keras you can install both Keras and the GPU version of TensorFlow with:

[library](https://rdrr.io/r/base/library.html)(keras)

[install\_keras](https://tensorflow.rstudio.com/keras/reference/install_keras.html)(tensorflow = "gpu")

Note that on all platforms you must be running an NVIDIA® GPU with CUDA® Compute Capability 3.5 or higher in order to run the GPU version of TensorFlow.

Prerequisties

WINDOWS

This article describes how to detect whether your graphics card uses an NVIDIA® GPU:

Once you’ve confirmed that you have an NVIDIA® GPU, the following article describes how to install required software components including the CUDA Toolkit v10.0, required NVIDIA® drivers, and cuDNN >= v7.4.1:

Note that the documentation on installation of the last component (cuDNN v7.4.1) is a bit sparse. Once you join the NVIDIA® developer program and download the zip file containing cuDNN you need to extract the zip file and add the location where you extracted it to your system PATH.

UBUNTU

This article describes how to install required software components including the CUDA Toolkit v10.0, required NVIDIA® drivers, and cuDNN >= v7.4.1:

The specifics of installing required software differ by Linux version so please review the NVIDIA® documentation carefully to ensure you install everything correctly.

The following section provides as example of the installation commands you might use on Ubuntu 16.04.

UBUNTU 16.04 EXAMPLE

First, install the NVIDIA drivers:

# Add NVIDIA package repositories

# Add HTTPS support for apt-key

sudo apt-get install gnupg-curl

wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86\_64/cuda-repo-ubuntu1604\_10.0.130-1\_amd64.deb

sudo dpkg -i cuda-repo-ubuntu1604\_10.0.130-1\_amd64.deb

sudo apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86\_64/7fa2af80.pub

sudo apt-get update

wget http://developer.download.nvidia.com/compute/machine-learning/repos/ubuntu1604/x86\_64/nvidia-machine-learning-repo-ubuntu1604\_1.0.0-1\_amd64.deb

sudo apt install ./nvidia-machine-learning-repo-ubuntu1604\_1.0.0-1\_amd64.deb

sudo apt-get update

# Install NVIDIA driver

# Issue with driver install requires creating /usr/lib/nvidia

sudo mkdir /usr/lib/nvidia

sudo apt-get install --no-install-recommends nvidia-410

# Reboot. Check that GPUs are visible using the command: nvidia-smi

Next install CUDA Toolkit v10.0 and cuDNN v7.4.1 with:

# Install development and runtime libraries (~4GB)

sudo apt-get install --no-install-recommends \

cuda-10-0 \

libcudnn7=7.4.1.5-1+cuda10.0 \

libcudnn7-dev=7.4.1.5-1+cuda10.0

Note that it’s important to download CUDA 10.0 (rather than CUDA 10.1, which may be the choice initially presented) as v10.0 is what TensorFlow is built against.

You can see more for the installation [here](https://www.tensorflow.org/install/gpu#linux_setup).

ENVIRONMENT VARIABLES

On Linux, part of the setup for CUDA libraries is adding the path to the CUDA binaries to your PATH and LD\_LIBRARY\_PATH as well as setting the CUDA\_HOME environment variable. You will set these variables in distinct ways depending on whether you are installing TensorFlow on a single-user workstation or on a multi-user server. If you are running RStudio Server there is some additional setup required which is also covered below.

In all cases these are the environment variables that need to be set/modified in order for TensorFlow to find the required CUDA libraries. For example (paths will change depending on your specific installation of CUDA):

export CUDA\_HOME=/usr/local/cuda

export LD\_LIBRARY\_PATH=${LD\_LIBRARY\_PATH}:${CUDA\_HOME}/lib64

PATH=${CUDA\_HOME}/bin:${PATH}

export PATH

SINGLE-USER INSTALLATION

In a single-user environment (e.g. a desktop system) you should define the environment variables within your ~/.profile file. It’s necessary to use ~/.profile rather than ~/.bashrc, because ~/.profile is read by desktop applications (e.g. RStudio) as well as terminal sessions whereas ~/.bashrc applies only to terminal sessions.

Note that you need to restart your system after editing the ~/.profile file for the changes to take effect. Note also that the ~/.profile file will not be read by bash if you have either a ~/.bash\_profile or ~/.bash\_login file.

To summarize the recommendations above:

* Define CUDA related environment variables in ~/.profile rather than ~/.bashrc;
* Ensure that you don’t have either a ~/.bash\_profile or ~/.bash\_login file (as these will prevent bash from seeing the variables you’ve added into ~/.profile);
* Restart your system after editing ~/.profile so that the changes take effect.

MULTI-USER INSTALLATION

In a multi-user installation (e.g. a server) you should define the environment variables within the system-wide bash startup file (/etc/profile) so all users have access to them.

If you are running RStudio Server you need to also provide these variable definitions in an R / RStudio specific fashion (as RStudio Server doesn’t execute system profile scripts for R sessions).

To modify the LD\_LIBRARY\_PATH you use the rsession-ld-library-path in the /etc/rstudio/rserver.conf configuration file

*/etc/rstudio/rserver.conf*

rsession-ld-library-path=/usr/local/cuda/lib64:/usr/local/cuda/extras/CUPTI/lib64

You should set the CUDA\_HOME and PATH variables in the /usr/lib/R/etc/Rprofile.site configuration file:

*/usr/lib/R/etc/Rprofile.site*

[Sys.setenv](https://rdrr.io/r/base/Sys.setenv.html)(CUDA\_HOME="/usr/local/cuda")

[Sys.setenv](https://rdrr.io/r/base/Sys.setenv.html)(PATH=[paste](https://rdrr.io/r/base/paste.html)([Sys.getenv](https://rdrr.io/r/base/Sys.getenv.html)("PATH"), "/usr/local/cuda/bin", sep = ":"))

In a server environment you might also find it more convenient to install TensorFlow into a system-wide location where all users of the server can share access to it.

MAC OS X

As of version 1.2 of TensorFlow, GPU support is no longer available on Mac OS X. If you want to use a GPU on Mac OS X you will need to install TensorFlow v1.1 as follows:

[library](https://rdrr.io/r/base/library.html)(tensorflow)

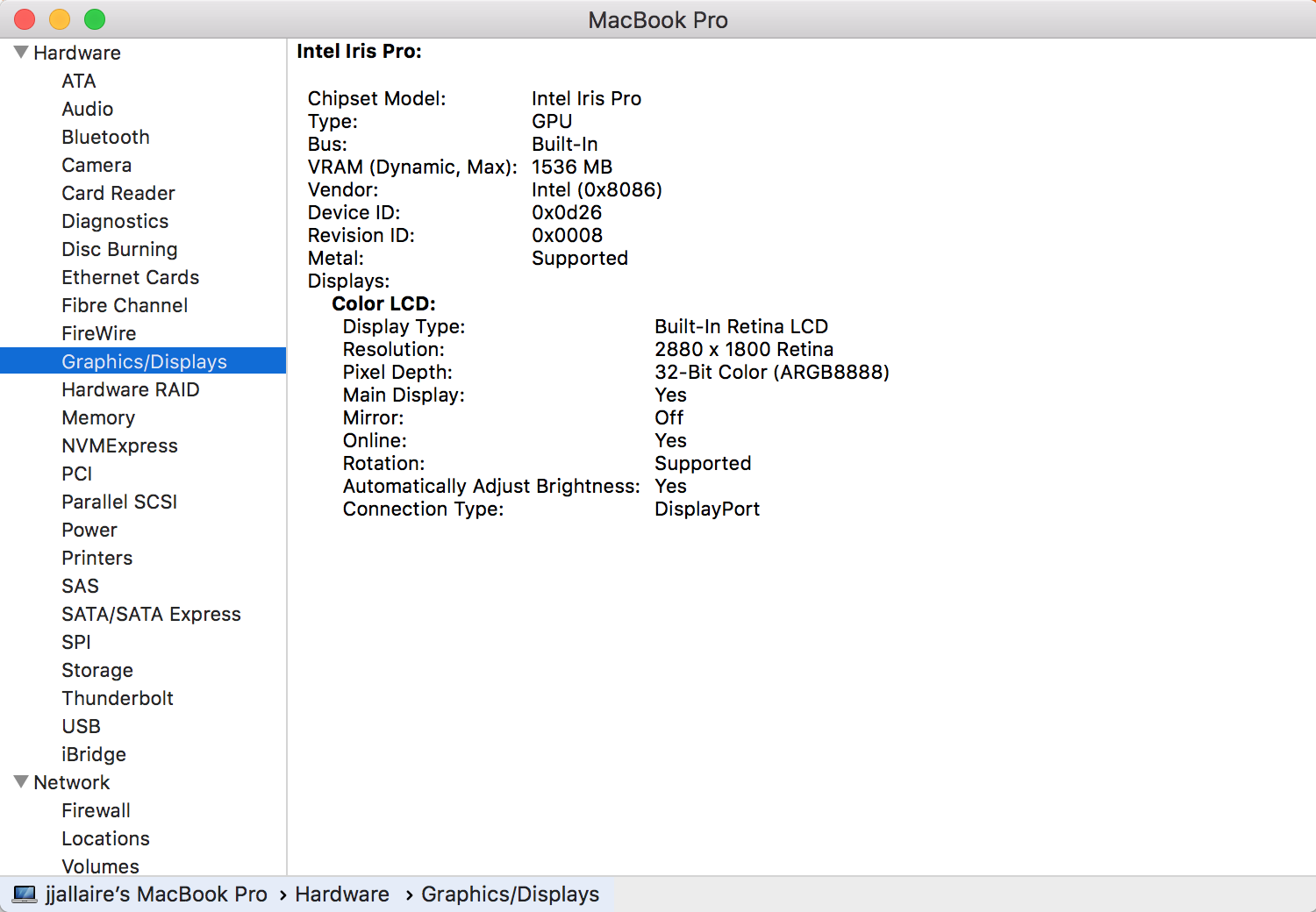
[install\_tensorflow](https://tensorflow.rstudio.com/tensorflow/reference/install_tensorflow.html)(version = "1.1-gpu")

However, before you install you should ensure that you have an NVIDIA® GPU and that you have the required CUDA libraries on your system.

While some older Macs include NVIDIA® GPU’s, most Macs (especially newer ones) do not, so you should check the type of graphics card you have in your Mac before proceeding.

Here is a list of Mac systems which include built in NVIDIA GPU’s:

You can check which graphics card your Mac has via the System Report button found within the About This Mac dialog:



The MacBook Pro system displayed above *does not* have an NVIDIA® GPU installed (rather it has an Intel Iris Pro).

If you do have an NVIDIA® GPU, the following article describes how to install the base CUDA libraries:

You also need to intall the cuDNN library 5.1 library for OS X from here:

After installing these components, you need to ensure that both CUDA and cuDNN are available to your R session via the DYLD\_LIBRARY\_PATH. This typically involves setting environment variables in your .bash\_profile as described in the NVIDIA documentation for CUDA and cuDNN.

Note that environment variables set in .bash\_profile will not be available by default to OS X desktop applications like R GUI and RStudio. To use CUDA within those environments you should start the application from a system terminal as follows:

open -a R # R GUI

open -a RStudio # RStudio

Installation

SINGLE USER

In a single-user desktop environment you can install TensorFlow with GPU support via:

[library](https://rdrr.io/r/base/library.html)(tensorflow)

[install\_tensorflow](https://tensorflow.rstudio.com/tensorflow/reference/install_tensorflow.html)(version = "gpu")

If this version doesn’t load successfully you should review the prerequisites above and ensure that you’ve provided definitions of CUDA environment variables as recommended above.

(e.g. virtualenv vs. conda installation, installing development versions, etc.).

MULTIPLE USERS

In a multi-user server environment you may want to install a system-wide version of TensorFlow with GPU support so all users can share the same configuration. To do this, start by following the directions for native pip installation of the GPU version of TensorFlow here:

There are some components of TensorFlow (e.g. the [Keras](https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/keras) library) which have dependencies on additional Python packages.

You can install Keras and it’s optional dependencies with the following command (ensuring you have the correct privilege to write to system library locations as required via sudo, etc.):

pip install keras h5py pyyaml requests Pillow scipy

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!