Loading libraries

```
# load libraries
library(tidyverse)
library(tensorflow)
library(keras)
```

```
tf$random$set_seed(42)

# check TF version
tf_version()
#[1] '2.2'

# check if keras is available
is_keras_available()
#[1] TRUE
```

Loading images (data)

The dataset I am using here is the fruit images dataset from Kaggle. I downloaded it to my computer and unpacked it. Because I don't want to build a model for all the different fruits, I define a list of fruits (corresponding to the folder names) that I want to include in the model.

```
# path to image folders
train_image_files_path <- "/fruits/Training/"

# list of fruits to modle
fruit_list <- c("Kiwi", "Banana", "Apricot", "Avocado", "Cocos", "Clementine",
"Mandarine", "Orange",
                "Limes", "Lemon", "Peach", "Plum", "Raspberry", "Strawberry",
"Pineapple", "Pomegranate")

# number of output classes (i.e. fruits)
output_n <- length(fruit_list)

# image size to scale down to (original images are 100 x 100 px)
img_width <- 20
img_height <- 20
target_size <- c(img_width, img_height)

# RGB = 3 channels
channels <- 3

# define batch size
batch_size <- 32
```

The handy `image_data_generator()` and `flow_images_from_directory()` functions can be used to load images from a directory without having to store all data in memory at the same time. Instead `image_data_generator` will loop over the data and process the images in batches.

When we train our model, we want to have a way to judge how well it learned and if learning improves over the epochs. Therefore, we want to use validation data, to make these performance measures less biased compared to using the training data only. In Keras, we can either give a specific **validation set** (as I did in the old article on image classification with Keras) or we define a validation split.

If you want to use data augmentation, you can directly define how and in what way you want to augment your images with `image_data_generator`. Here I am not augmenting the data, I only scale the pixel values to fall between 0 and 1.

```
train_data_gen <- image_data_generator(
  rescale = 1/255,
  validation_split = 0.3)
```

Now we load the images into memory and resize them.

```
# training images
train_image_array_gen <- flow_images_from_directory(train_image_files_path,
                                          train_data_gen,
                                          subset = 'training',
                                          target_size = target_size,
                                          class_mode = "categorical",
                                          classes = fruit_list,
                                          batch_size = batch_size,
                                          seed = 42)
#Found 5401 images belonging to 16 classes.

# validation images
valid_image_array_gen <- flow_images_from_directory(train_image_files_path,
                                          train_data_gen,
                                          subset = 'validation',
                                          target_size = target_size,
                                          class_mode = "categorical",
                                          classes = fruit_list,
                                          batch_size = batch_size,
                                          seed = 42)
#Found 2308 images belonging to 16 classes.

cat("Number of images per class:")
table(factor(train_image_array_gen$classes))
#Number of images per class:
#  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
#327 343 345 299 343 343 343 336 343 345 345 313 343 345 343 345
```

Note, that even though Keras' `image_data_generator` recognizes folder names as class labels for classification tasks, these labels will be converted into indices (alphabetical starting from 0) for training and prediction later. Thus, it is useful to create a library object that matches these indices back to human-interpretable labels.

```
train_image_array_gen_t <- train_image_array_gen$class_indices %>%
  as.tibble()

cat("\nClass label vs index mapping:\n")

##
## Class label vs index mapping:

train_image_array_gen_t

## # A tibble: 1 x 16
##    Kiwi Banana Apricot Avocado Cocos Clementine Mandarine Orange Limes Lemon
##
## 1     0      1       2       3     4          5         6      7     8     9
## # … with 6 more variables: Peach , Plum , Raspberry ,
## #   Strawberry , Pineapple , Pomegranate
```

Training the model

Now, I define and train the model just as before:

```
# number of training samples
train_samples <- train_image_array_gen$n
# number of validation samples
valid_samples <- valid_image_array_gen$n

# define number of epochs
epochs <- 10

# initialise model
model <- keras_model_sequential()

# add layers
model %>%
  layer_conv_2d(filter = 32, kernel_size = c(3,3), padding = "same", input_shape
= c(img_width, img_height, channels)) %>%
  layer_activation("relu") %>%

  # Second hidden layer
  layer_conv_2d(filter = 16, kernel_size = c(3,3), padding = "same") %>%
  layer_activation_leaky_relu(0.5) %>%
  layer_batch_normalization() %>%

  # Use max pooling
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(0.25) %>%

  # Flatten max filtered output into feature vector
  # and feed into dense layer
  layer_flatten() %>%
  layer_dense(100) %>%
  layer_activation("relu") %>%
  layer_dropout(0.5) %>%

  # Outputs from dense layer are projected onto output layer
  layer_dense(output_n) %>%
  layer_activation("softmax")

# compile
model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_rmsprop(lr = 0.0001, decay = 1e-6),
  metrics = "accuracy"
)
```

Fitting the model: because I used `image_data_generator()` and `flow_images_from_directory()` I am now also using the `fit_generator()` to run the training.

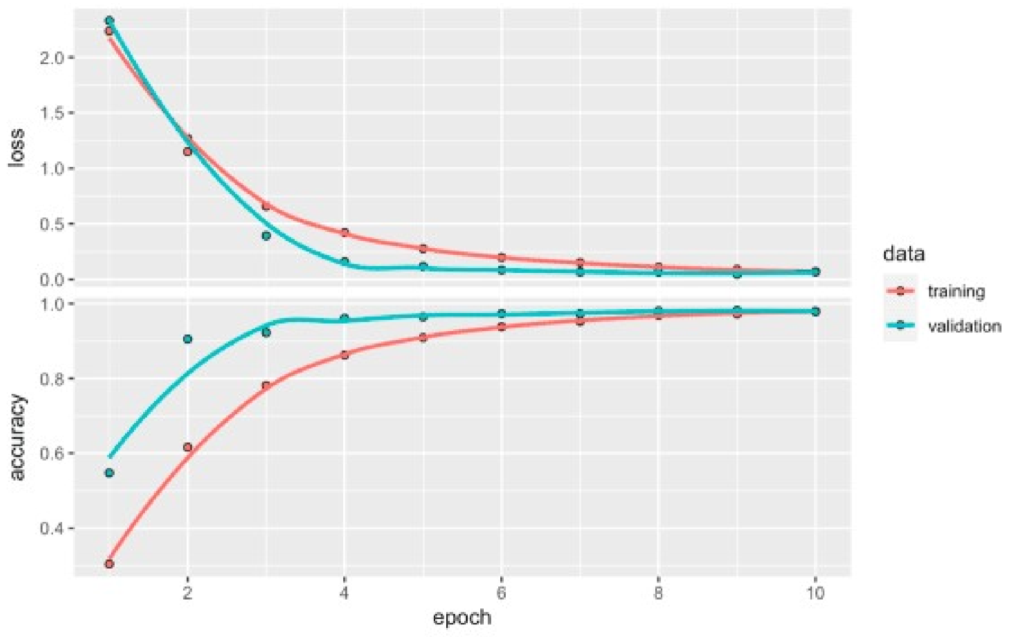
Note: In future releases of TensorFlow, generator functions will be deprecated (i.e. `fit_generator()`, `evaluate_generator()` & `predict_generator()`) and as of TensorFlow 2.1.0 the regular functions (`fit()`, `evaluate()` & `predict()`) handle generators directly. However, this does not seem to be implemented in the Keras (R-) package yet (see `?fit.keras.engine.training.Model`), so I'll be sticking to the old way of doing things for now.

```
# fit
hist <- model %>% fit_generator(
  # training data
  train_image_array_gen,
  # epochs
  steps_per_epoch = as.integer(train_samples / batch_size),
  epochs = epochs,

  # validation data
  validation_data = valid_image_array_gen,
  validation_steps = as.integer(valid_samples / batch_size)
)
```

```
Epoch 1/10
168/168 [==============================] - 8s 51ms/step - loss: 1.8614 -
accuracy: 0.4194 - val_loss: 2.1271 - val_accuracy: 0.6897
Epoch 2/10
168/168 [==============================] - 6s 37ms/step - loss: 0.7967 -
accuracy: 0.7463 - val_loss: 0.9940 - val_accuracy: 0.9444
Epoch 3/10
168/168 [==============================] - 6s 34ms/step - loss: 0.4556 -
accuracy: 0.8622 - val_loss: 0.2636 - val_accuracy: 0.9722
Epoch 4/10
168/168 [==============================] - 6s 37ms/step - loss: 0.3001 -
accuracy: 0.9074 - val_loss: 0.1298 - val_accuracy: 0.9779
Epoch 5/10
168/168 [==============================] - 6s 35ms/step - loss: 0.2010 -
accuracy: 0.9417 - val_loss: 0.1194 - val_accuracy: 0.9757
Epoch 6/10
168/168 [==============================] - 6s 34ms/step - loss: 0.1539 -
accuracy: 0.9581 - val_loss: 0.0768 - val_accuracy: 0.9688
Epoch 7/10
168/168 [==============================] - 6s 36ms/step - loss: 0.1147 -
accuracy: 0.9672 - val_loss: 0.0574 - val_accuracy: 0.9831
Epoch 8/10
168/168 [==============================] - 6s 34ms/step - loss: 0.0831 -
accuracy: 0.9771 - val_loss: 0.0676 - val_accuracy: 0.9826
Epoch 9/10
168/168 [==============================] - 6s 35ms/step - loss: 0.0713 -
accuracy: 0.9786 - val_loss: 0.0542 - val_accuracy: 0.9796
Epoch 10/10
168/168 [==============================] - 6s 35ms/step - loss: 0.0572 -
accuracy: 0.9842 - val_loss: 0.0548 - val_accuracy: 0.9835
```

In RStudio we are seeing the output as an interactive plot in the “Viewer” pane but we can also plot it:

```
plot(hist)
```



```
model %>% save_model_hdf5("my_model.h5")

model <- load_model_hdf5("my_model.h5")
```

Predicting on “new” set of images

Here is what’s new: how to use the model we just trained for predicting on new images. Alternatively, load a previously trained model and use that for predictions. Just keep in mind, that these “new” images here aren’t actually new. They come from the same data source, i.e. the same distribution and thus aren’t independent enough to give an accurate assessment of how well the model generalizes.

The easiest and most efficient way to predict on a set of new images (and resize them to the target width and height), is to use the **image data generator** just as we did for reading in the training data above. Of course, you will need to know how the training images were preprocessed, so that you can apply the same steps to your test images. Here, the image pixel values were scaled to fall between 0 and 1 and the width/height were set to 20 pixels.

```
# path to image folders
test_image_files_path <- "/fruits/Test/"

test_datagen <- image_data_generator(rescale = 1/255)


test_generator <- flow_images_from_directory(
        test_image_files_path,
        test_datagen,
        target_size = target_size,
        class_mode = "categorical",
        classes = fruit_list,
        batch_size = 1,
        shuffle = FALSE,
        seed = 42)
#Found 2592 images belonging to 16 classes.
```

Next, we can use the image data generator we created for our test images on two functions. First, I’ll run `evaluate_generator`, to get an overall idea of how well the model predicted the test images (note, that this requires for us to know the correct predictions for our images):

```
model %>%
  evaluate_generator(test_generator,
                     steps = as.integer(test_generator$n))
```

```
      loss   accuracy
0.02915302 0.99305558
```

Usually, what's wanted is to get individual predictions for each test image. For that, we use the `predict_generator()` function. This function will return a matrix:

- each row represents one test image
- columns gives prediction probabilities for all possibles classes (if you are running a classifier)

In order to compare predictions with actual class labels, I am preparing the true labels so that they can be merged with the predictions:

```
classes <- test_generator$classes %>%
  factor() %>%
  table() %>%
  as.tibble()
colnames(classes)[1] <- "value"

# create library of indices & class labels
indices <- test_generator$class_indices %>%
  as.data.frame() %>%
  gather() %>%
  mutate(value = as.character(value)) %>%
  left_join(classes, by = "value")
```

Now, I'm running the predictions, rename the columns to show the class labels instead of indices and I am merging the true labels, together with their corresponding labels and the total number of images in each class.

(Running `test_generator$reset()` isn't strictly necessary, but sometimes strange things happen if you are using the image data generator several times, so I always run reset first, to be on the safe side.)

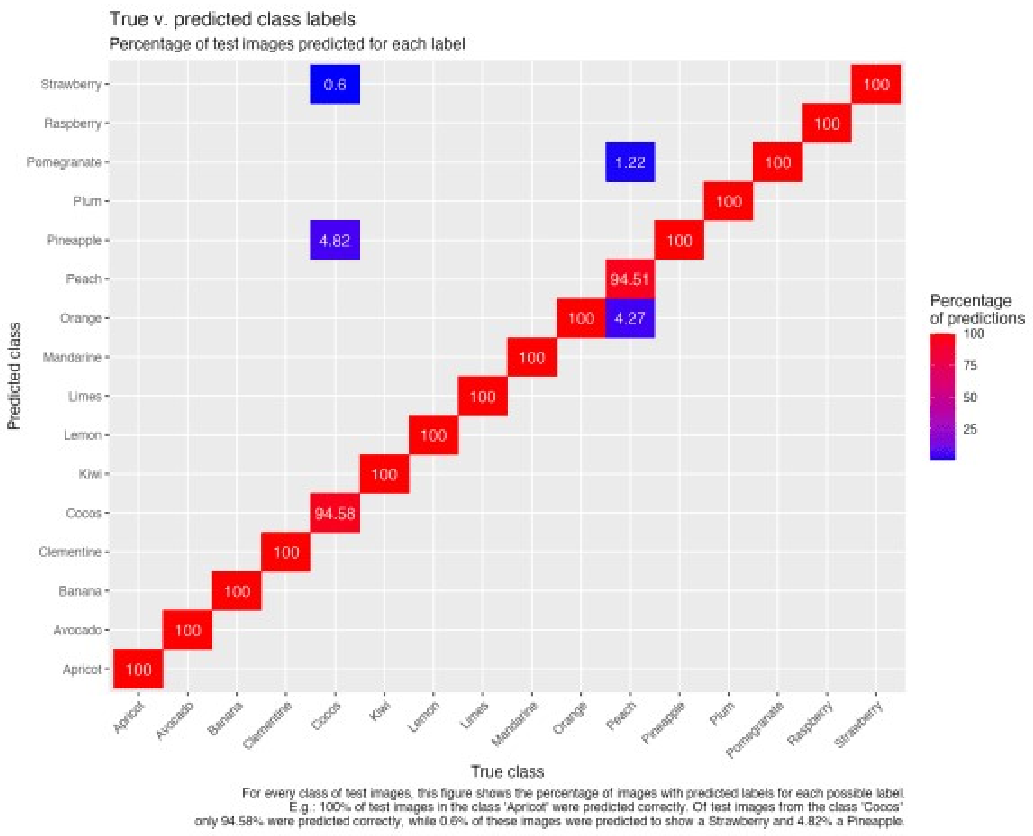
```
# predict on test data
test_generator$reset()
predictions <- model %>%
  predict_generator(
    generator = test_generator,
    steps = as.integer(test_generator$n)
    ) %>%
  round(digits = 2) %>%
  as.tibble()

colnames(predictions) <- indices$key

predictions <- predictions %>%
  mutate(truth_idx = as.character(test_generator$classes)) %>%
  left_join(indices, by = c("truth_idx" = "value"))
```

If you want to know which class is predicted for each image, you need to filter for the class with the highest prediction probability (here, tidyverse-style). Afterwards, for each class, I am also counting how many images were predicted with which label.

```
pred_analysis <- predictions %>%
  mutate(img_id = seq(1:test_generator$n)) %>%
  gather(pred_lbl, y, Kiwi:Pomegranate) %>%
  group_by(img_id) %>%
  filter(y == max(y)) %>%
  arrange(img_id) %>%
  group_by(key, n, pred_lbl) %>%
  count()
```

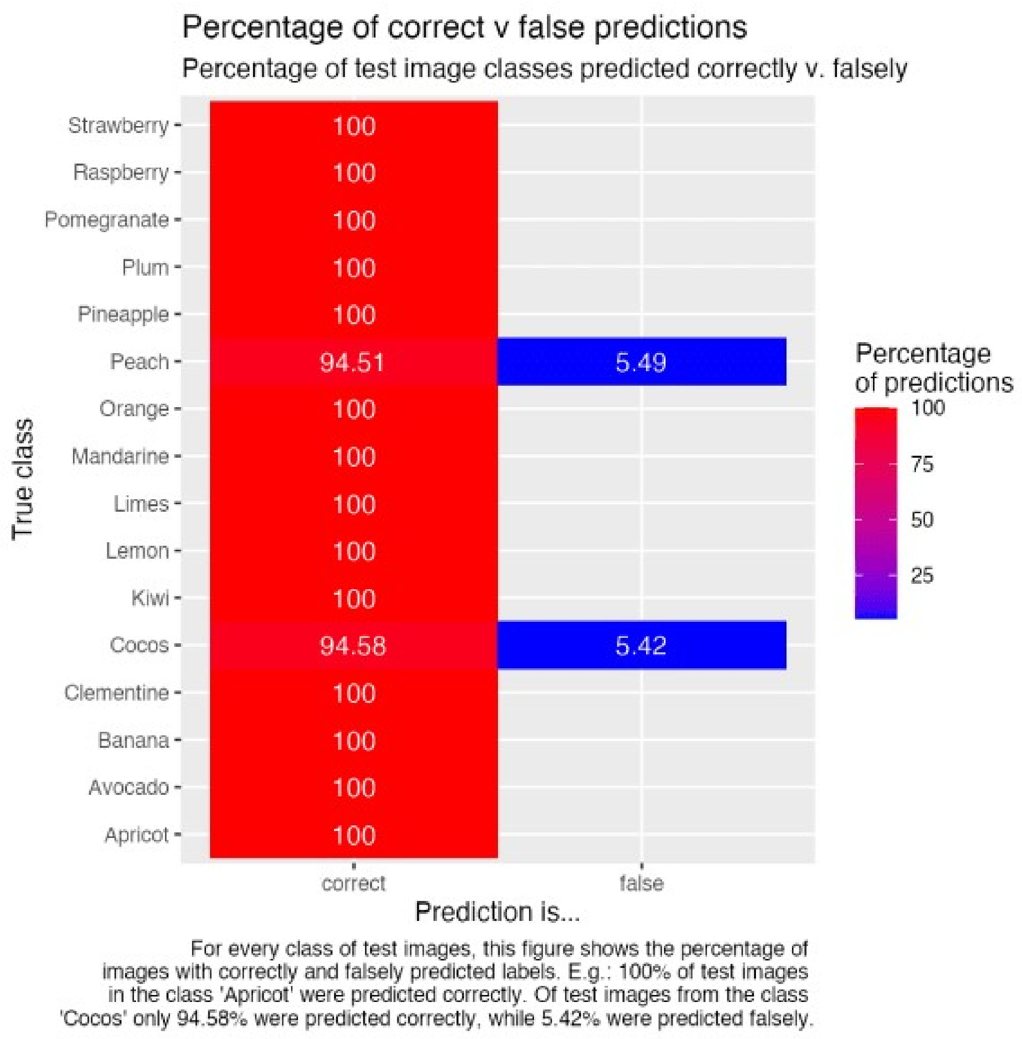
And since plotting makes comparing much easier than looking at a table, I'm creating a tile plot that shows the percentages of test images predicted for each label:

```
p <- pred_analysis %>%
  mutate(percentage_pred = nn / n * 100) %>%
  ggplot(aes(x = key, y = pred_lbl,
             fill = percentage_pred,
             label = round(percentage_pred, 2))) +
    geom_tile() +
    scale_fill_continuous() +
    scale_fill_gradient(low = "blue", high = "red") +
    geom_text(color = "white") +
    theme(axis.text.x = element_text(angle = 45, vjust = 1, hjust = 1)) +
    labs(x = "True class",
         y = "Predicted class",
         fill = "Percentage\nof predictions",
        title = "True v. predicted class labels",
        subtitle = "Percentage of test images predicted for each label",
        caption = "For every class of test images, this figure shows the
percentage of images with predicted labels for each possible label.
        E.g.: 100% of test images in the class 'Apricot' were predicted
correctly. Of test images from the class 'Cocos'
        only 94.58% were predicted correctly, while 0.6% of these images were
predicted to show a Strawberry and 4.82% a Pineapple.")
p
```



And another tile plot showing the percentages of correct and falsely predicted images in each class:

```
p2 <- pred_analysis %>%
  mutate(prediction = case_when(
    key == pred_lbl ~ "correct",
    TRUE ~ "false"
  )) %>%
  group_by(key, prediction, n) %>%
  summarise(sum = sum(nn)) %>%
  mutate(percentage_pred = sum / n * 100) %>%
  ggplot(aes(x = key, y = prediction,
             fill = percentage_pred,
             label = round(percentage_pred, 2))) +
    geom_tile() +
    scale_fill_continuous() +
    geom_text(color = "white") +
    coord_flip() +
    scale_fill_gradient(low = "blue", high = "red") +
    labs(x = "True class",
         y = "Prediction is...",
         fill = "Percentage\nof predictions",
        title = "Percentage of correct v false predictions",
        subtitle = "Percentage of test image classes predicted correctly v.
falsely",
        caption = "For every class of test images, this figure shows the
percentage of
        images with correctly and falsely predicted labels. E.g.: 100% of test
images
        in the class 'Apricot' were predicted correctly. Of test images from the
class
        'Cocos' only 94.58% were predicted correctly, while 5.42% were predicted
falsely.")
p2
```



As we can see, the model is quite accurate on the test data. However, we need to keep in mind that our images are very uniform, they all have the same white background and show the fruits centered and without anything else in the images. Thus, our model will not work with images that don’t look similar as the ones we trained on (that’s also why we can achieve such good results with such a small neural net).

```
devtools::session_info()
```

```
## ─ Session info ────────────────────────────────────────────────────────────
───
##  setting  value
##  version  R version 4.0.2 (2020-06-22)
##  os       macOS Catalina 10.15.6
##  system   x86_64, darwin17.0
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       Europe/Berlin
##  date     2020-09-14
##
## ─ Packages ────────────────────────────────────────────────────────────
───────
##  package     * version date       lib source
##  assertthat    0.2.1   2019-03-21 [1] CRAN (R 4.0.0)
##  backports     1.1.9   2020-08-24 [1] CRAN (R 4.0.2)
##  base64enc     0.1-3   2015-07-28 [1] CRAN (R 4.0.0)
##  blob          1.2.1   2020-01-20 [1] CRAN (R 4.0.2)
##  blogdown      0.20.1  2020-09-09 [1] Github (rstudio/blogdown@d96fe78)
##  bookdown      0.20    2020-06-23 [1] CRAN (R 4.0.2)
##  broom         0.7.0   2020-07-09 [1] CRAN (R 4.0.2)
##  callr         3.4.4   2020-09-07 [1] CRAN (R 4.0.2)
##  cellranger    1.1.0   2016-07-27 [1] CRAN (R 4.0.0)
##  cli           2.0.2   2020-02-28 [1] CRAN (R 4.0.0)
##  colorspace    1.4-1   2019-03-18 [1] CRAN (R 4.0.0)
##  crayon        1.3.4   2017-09-16 [1] CRAN (R 4.0.0)
##  DBI           1.1.0   2019-12-15 [1] CRAN (R 4.0.0)
##  dbplyr        1.4.4   2020-05-27 [1] CRAN (R 4.0.2)
##  desc          1.2.0   2018-05-01 [1] CRAN (R 4.0.0)
##  devtools      2.3.1   2020-07-21 [1] CRAN (R 4.0.2)
##  digest        0.6.25  2020-02-23 [1] CRAN (R 4.0.0)
##  dplyr       * 1.0.2   2020-08-18 [1] CRAN (R 4.0.2)
##  ellipsis      0.3.1   2020-05-15 [1] CRAN (R 4.0.0)
##  evaluate      0.14    2019-05-28 [1] CRAN (R 4.0.1)
##  fansi         0.4.1   2020-01-08 [1] CRAN (R 4.0.0)
##  forcats     * 0.5.0   2020-03-01 [1] CRAN (R 4.0.0)
##  fs            1.5.0   2020-07-31 [1] CRAN (R 4.0.2)
##  generics      0.0.2   2018-11-29 [1] CRAN (R 4.0.0)
##  ggplot2     * 3.3.2   2020-06-19 [1] CRAN (R 4.0.2)
##  glue          1.4.2   2020-08-27 [1] CRAN (R 4.0.2)
##  gtable        0.3.0   2019-03-25 [1] CRAN (R 4.0.0)
##  haven         2.3.1   2020-06-01 [1] CRAN (R 4.0.2)
##  hms           0.5.3   2020-01-08 [1] CRAN (R 4.0.0)
##  htmltools     0.5.0   2020-06-16 [1] CRAN (R 4.0.2)
##  httr          1.4.2   2020-07-20 [1] CRAN (R 4.0.2)
##  jsonlite      1.7.1   2020-09-07 [1] CRAN (R 4.0.2)
##  keras       * 2.3.0.0 2020-05-19 [1] CRAN (R 4.0.2)
##  knitr         1.29    2020-06-23 [1] CRAN (R 4.0.2)
##  lattice       0.20-41 2020-04-02 [1] CRAN (R 4.0.2)
##  lifecycle     0.2.0   2020-03-06 [1] CRAN (R 4.0.0)
##  lubridate     1.7.9   2020-06-08 [1] CRAN (R 4.0.2)
##  magrittr      1.5     2014-11-22 [1] CRAN (R 4.0.0)
##  Matrix        1.2-18  2019-11-27 [1] CRAN (R 4.0.2)
##  memoise       1.1.0   2017-04-21 [1] CRAN (R 4.0.0)
##  modelr        0.1.8   2020-05-19 [1] CRAN (R 4.0.2)
##  munsell       0.5.0   2018-06-12 [1] CRAN (R 4.0.0)
##  pillar        1.4.6   2020-07-10 [1] CRAN (R 4.0.2)
##  pkgbuild      1.1.0   2020-07-13 [1] CRAN (R 4.0.2)
##  pkgconfig     2.0.3   2019-09-22 [1] CRAN (R 4.0.0)
##  pkgload       1.1.0   2020-05-29 [1] CRAN (R 4.0.2)
##  prettyunits   1.1.1   2020-01-24 [1] CRAN (R 4.0.0)
##  processx      3.4.4   2020-09-03 [1] CRAN (R 4.0.2)
##  ps            1.3.4   2020-08-11 [1] CRAN (R 4.0.2)
##  purrr       * 0.3.4   2020-04-17 [1] CRAN (R 4.0.0)
##  R6            2.4.1   2019-11-12 [1] CRAN (R 4.0.0)
##  Rcpp          1.0.5   2020-07-06 [1] CRAN (R 4.0.2)
##  readr       * 1.3.1   2018-12-21 [1] CRAN (R 4.0.0)
##  readxl        1.3.1   2019-03-13 [1] CRAN (R 4.0.0)
##  remotes       2.2.0   2020-07-21 [1] CRAN (R 4.0.2)
##  reprex        0.3.0   2019-05-16 [1] CRAN (R 4.0.0)
##  reticulate    1.16    2020-05-27 [1] CRAN (R 4.0.2)
##  rlang         0.4.7   2020-07-09 [1] CRAN (R 4.0.2)
##  rmarkdown     2.3     2020-06-18 [1] CRAN (R 4.0.2)
##  rprojroot     1.3-2   2018-01-03 [1] CRAN (R 4.0.0)
##  rstudioapi    0.11    2020-02-07 [1] CRAN (R 4.0.0)
##  rvest         0.3.6   2020-07-25 [1] CRAN (R 4.0.2)
##  scales        1.1.1   2020-05-11 [1] CRAN (R 4.0.0)
##  sessioninfo   1.1.1   2018-11-05 [1] CRAN (R 4.0.0)
##  stringi       1.5.3   2020-09-09 [1] CRAN (R 4.0.2)
##  stringr     * 1.4.0   2019-02-10 [1] CRAN (R 4.0.0)
##  tensorflow  * 2.2.0   2020-05-11 [1] CRAN (R 4.0.0)
##  testthat      2.3.2   2020-03-02 [1] CRAN (R 4.0.0)
##  tfruns        1.4     2018-08-25 [1] CRAN (R 4.0.0)
##  tibble      * 3.0.3   2020-07-10 [1] CRAN (R 4.0.2)
##  tidyr       * 1.1.2   2020-08-27 [1] CRAN (R 4.0.2)
##  tidyselect    1.1.0   2020-05-11 [1] CRAN (R 4.0.0)
##  tidyverse   * 1.3.0   2019-11-21 [1] CRAN (R 4.0.0)
##  usethis       1.6.1   2020-04-29 [1] CRAN (R 4.0.0)
##  utf8          1.1.4   2018-05-24 [1] CRAN (R 4.0.0)
##  vctrs         0.3.4   2020-08-29 [1] CRAN (R 4.0.2)
##  whisker       0.4     2019-08-28 [1] CRAN (R 4.0.0)
##  withr         2.2.0   2020-04-20 [1] CRAN (R 4.0.0)
##  xfun          0.17    2020-09-09 [1] CRAN (R 4.0.2)
##  xml2          1.3.2   2020-04-23 [1] CRAN (R 4.0.0)
##  yaml          2.2.1   2020-02-01 [1] CRAN (R 4.0.0)
##  zeallot       0.1.0   2018-01-28 [1] CRAN (R 4.0.0)
##
## [1] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
```