Loading libraries

```
# load libraries
library(tidyverse)
library(tidymodels)
library(tensorflow)
library(keras)
library(deepviz) # https://github.com/andrie/deepviz


tf$random$set_seed(42)

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

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

Sequential models

For many models, `keras_model_sequential` is sufficient! Whenever your model has one input (i.e. one set of images, one matrix, one set of texts, etc.), one output layer and a linear order of layers in between, you can use the `keras_model_sequential()` function. For example, it can be used to build simple MLPs, CNNs and (Bidirectional) LSTMs.

You can find a full working example here.

Complex models

So, when would you use the **Functional API**? You can't use the `keras_model_sequential` when you want to build a more complex model, e.g. in case you have multiple (types of) inputs (like matrix, images and text) or multiple outputs, or even complex connections between layers that can't be described in a linear fashion (like directed acyclic graphs, e.g. in inception modules or residual blocks). Examples include an Auxiliary Classifier Generative Adversarial Network (ACGAN) and neural style transfer.

How to use the Functional API

The main function for using the **Functional API** is called `keras_model()`. With `keras_model`, you combine input and output layers. To make it easier to understand, let's look at a simple example. Below, I'll be building the same model from last week's blogpost, where I trained an image classification model with `keras_model_sequential`. Just that now, I'll be using the **Functional API** instead.

The first part is identical to before: defining the image data generator to read in the training images.

```
# path to image folders
```

```
train_image_files_path <- "/fruits/Training/"


# list of fruits to model
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


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


# 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.



# 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
```

Here's what's different with the **Functional API**:

With `keras_model_sequential`, we start with that function and add layers one after the other until we get to the output layer. The first layer after `keras_model_sequential()` needs to have input parameters matching the input data's dimensions (or you start with `layer_input()` as first layer). This complete model is then compiled and fit.

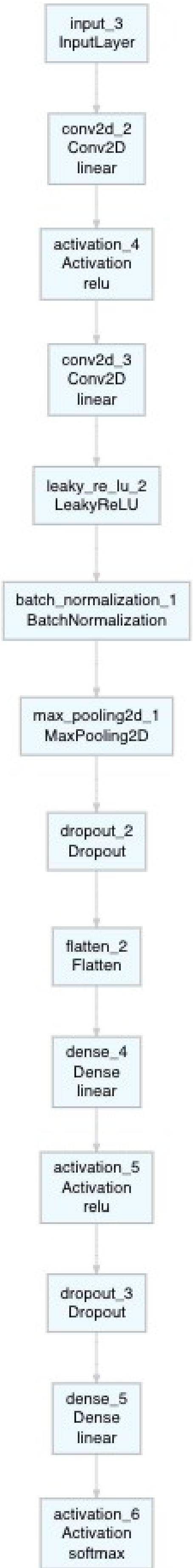
With the **Functional API**, we start by defining the input with `layer_input` as a separate object. At least one other object is defined containing additional layers and the output layer. With `keras_model()`, we then combine input and output into one model that's compiled and fit the same way a sequential model would be.

```
# input layer
inputs <- layer_input(shape = c(img_width, img_height, channels))

# outputs compose input + dense layers
predictions <- inputs %>%
  layer_conv_2d(filter = 32, kernel_size = c(3,3), padding = "same")
%>%
  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")

# create and compile model
model_func <- keras_model(inputs = inputs,
                          outputs = predictions)

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

This model here is very straightforward and could have been built just as easily with `keras_model_sequential`. A nice way to visualize our model architecture (particularly, when we are building complex models), is to use a plotting function (here from `deepviz`):

```
model_func %>% plot_model()
```


input_3
InputLayer
conv2d_2
Conv2D
linear
activation_4
Activation
relu
conv2d_3
Conv2D
linear
leaky_re_lu_2
LeakyReLU
batch_normalization_1
BatchNormalization
max_pooling2d_1
MaxPooling2D
dropout_2
Dropout
flatten_2
Flatten
dense_4
Dense
linear
activation_5
Activation
relu
dropout_3
Dropout
dense_5
Dense
linear
activation_6
Activation
softmax

```
# fit
```

```
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 [==============================] - 9s 55ms/step - loss: 1.9132
- accuracy: 0.3986 - val_loss: 2.1233 - val_accuracy: 0.6797
Epoch 2/10
168/168 [==============================] - 6s 36ms/step - loss: 0.8035
- accuracy: 0.7469 - val_loss: 0.9889 - val_accuracy: 0.9362
Epoch 3/10
168/168 [==============================] - 6s 35ms/step - loss: 0.4647
- accuracy: 0.8560 - val_loss: 0.2568 - val_accuracy: 0.9648
Epoch 4/10
168/168 [==============================] - 6s 34ms/step - loss: 0.2953
- accuracy: 0.9126 - val_loss: 0.1357 - val_accuracy: 0.9705
Epoch 5/10
168/168 [==============================] - 6s 35ms/step - loss: 0.2089
- accuracy: 0.9428 - val_loss: 0.0888 - val_accuracy: 0.9692
Epoch 6/10
168/168 [==============================] - 6s 35ms/step - loss: 0.1505
- accuracy: 0.9568 - val_loss: 0.1001 - val_accuracy: 0.9774
Epoch 7/10
168/168 [==============================] - 6s 35ms/step - loss: 0.1135
- accuracy: 0.9689 - val_loss: 0.0793 - val_accuracy: 0.9805
Epoch 8/10
168/168 [==============================] - 6s 34ms/step - loss: 0.0957
- accuracy: 0.9734 - val_loss: 0.0628 - val_accuracy: 0.9818
Epoch 9/10
168/168 [==============================] - 6s 35ms/step - loss: 0.0733
- accuracy: 0.9797 - val_loss: 0.0525 - val_accuracy: 0.9831
Epoch 10/10
168/168 [==============================] - 6s 34ms/step - loss: 0.0607
- accuracy: 0.9844 - val_loss: 0.0438 - val_accuracy: 0.9848
```

Inception module

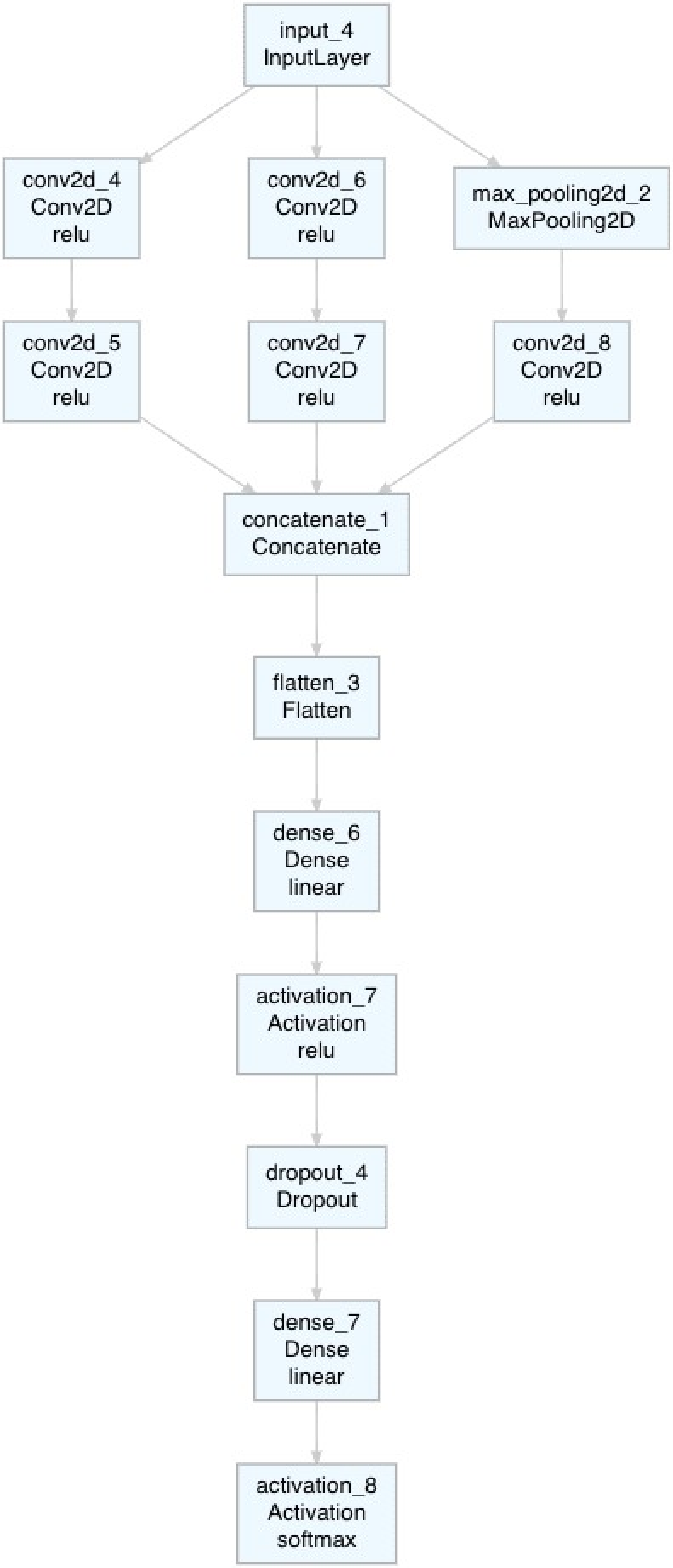
So, now that we've seen how the **Functional API** works in general, we can have a look at some more complex examples. The first one is the inception module. I'm not going to go into detail on what inception modules are and what they are useful for, but you can find a nice write-up here.

What's interesting in terms of building the model is that the inception module consists of **parallel**

(blocks of) layers. The output of an inception module is then combined back into one. In the model below, one input (our images from above) is being fed into one inception module consisting of three parallel blocks. These blocks can consist of any layers and layer combinations we want. The three different outputs are then combined back together by using `layer_concatenate()`. We could now create additional inception modules or we create the output layer. The rest is just as before: create model with `keras_model()`, compile and fit.

```
# input layer
inputs <- layer_input(shape = c(img_width, img_height, channels))

tower_1 <- inputs %>%
  layer_conv_2d(filters = 32, kernel_size = c(1, 1), padding='same',
activation='relu') %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), padding='same',
activation='relu')

tower_2 <- inputs %>%
  layer_conv_2d(filters = 32, kernel_size = c(1, 1), padding='same',
activation='relu') %>%
  layer_conv_2d(filters = 32, kernel_size = c(5, 5), padding='same',
activation='relu')

tower_3 <- inputs %>%
  layer_max_pooling_2d(pool_size = c(2, 2), strides = c(1, 1), padding
= 'same') %>%
  layer_conv_2d(filters = 32, kernel_size = c(1, 1), padding='same',
activation='relu')

output <- layer_concatenate(c(tower_1, tower_2, tower_3), axis = 1) %>%

  # 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")



# create and compile model
model_inc <- keras_model(inputs = inputs,
                     outputs = output) %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_rmsprop(lr = 0.0001, decay = 1e-6),
  metrics = "accuracy"
)
```

```
model_inc %>% plot_model()
```


input_4
InputLayer
conv2d_4
Conv2D
relu
conv2d_6
Conv2D
relu
max_pooling2d_2
MaxPooling2D
conv2d_5
Conv2D
relu
conv2d_7
Conv2D
relu
conv2d_8
Conv2D
relu
concatenate_1
Concatenate
flatten_3
Flatten
dense_6
Dense
linear
activation_7
Activation
relu
dropout_4
Dropout
dense_7
Dense
linear
activation_8
Activation
softmax

```
# fit
model_inc %>% 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 [==============================] - 16s 96ms/step - loss: 1.8178
- accuracy: 0.4377 - val_loss: 1.0520 - val_accuracy: 0.7630
Epoch 2/10
168/168 [==============================] - 17s 102ms/step - loss:
0.9190 - accuracy: 0.7283 - val_loss: 0.4448 - val_accuracy: 0.9271
Epoch 3/10
168/168 [==============================] - 16s 98ms/step - loss: 0.5711
- accuracy: 0.8374 - val_loss: 0.2429 - val_accuracy: 0.9531
Epoch 4/10
168/168 [==============================] - 17s 101ms/step - loss:
0.3920 - accuracy: 0.8931 - val_loss: 0.1571 - val_accuracy: 0.9644
Epoch 5/10
168/168 [==============================] - 19s 112ms/step - loss:
0.2787 - accuracy: 0.9201 - val_loss: 0.0967 - val_accuracy: 0.9657
Epoch 6/10
168/168 [==============================] - 17s 102ms/step - loss:
0.2191 - accuracy: 0.9398 - val_loss: 0.1057 - val_accuracy: 0.9653
Epoch 7/10
168/168 [==============================] - 16s 96ms/step - loss: 0.1828
- accuracy: 0.9438 - val_loss: 0.0658 - val_accuracy: 0.9922
Epoch 8/10
168/168 [==============================] - 16s 98ms/step - loss: 0.1463
- accuracy: 0.9590 - val_loss: 0.0536 - val_accuracy: 0.9852
Epoch 9/10
168/168 [==============================] - 17s 101ms/step - loss:
0.1266 - accuracy: 0.9616 - val_loss: 0.0520 - val_accuracy: 0.9913
Epoch 10/10
168/168 [==============================] - 16s 96ms/step - loss: 0.1040
- accuracy: 0.9687 - val_loss: 0.0526 - val_accuracy: 0.9822
```

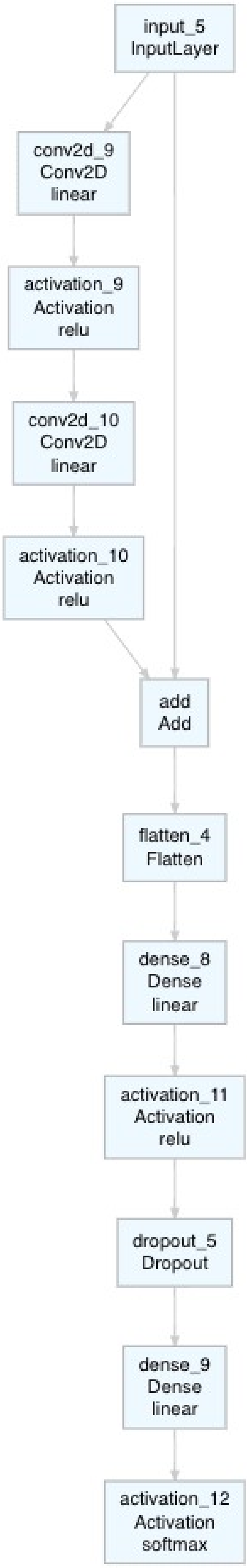
Residual blocks

Another useful model architecture uses residual blocks (residual neural networks, particularly ResNet). Residual blocks use so called skip connections. In my example below, the input follows two paths:

1. A convolution block that looks similar to the sequential model.
2. A skip connection that leaves out this convolution block and feeds the input straight to the next layer, which is following the convolution block.

We basically combine the original input back in with the convoluted output from the same input. If we want to combine two layers with the same dimensions, we can use `layer_add()`.

```
# input layer
inputs <- layer_input(shape = c(img_width, img_height, channels))

conv_block <- inputs %>%
  layer_conv_2d(filter = 32, kernel_size = c(3,3), padding = "same")
%>%
  layer_activation("relu") %>%

  # Second hidden layer
  layer_conv_2d(filter = 3, kernel_size = c(3,3), padding = "same") %>%
  layer_activation("relu")

output <- layer_add(c(inputs, conv_block)) %>%

  # 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")



# create and compile model
model_resid <- keras_model(inputs = inputs,
                     outputs = output) %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_rmsprop(lr = 0.0001, decay = 1e-6),
  metrics = "accuracy"
)



model_resid %>% plot_model()
```


input_5
InputLayer
conv2d_9
Conv2D
linear
activation_9
Activation
relu
conv2d_10
Conv2D
linear
activation_10
Activation
relu
add
Add
flatten_4
Flatten
dense_8
Dense
linear
activation_11
Activation
relu
dropout_5
Dropout
dense_9
Dense
linear
activation_12
Activation
softmax

```
# fit
```

```
model_resid %>% 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 [==============================] - 9s 51ms/step - loss: 2.3554
- accuracy: 0.2714 - val_loss: 1.9230 - val_accuracy: 0.5703
Epoch 2/10
168/168 [==============================] - 5s 30ms/step - loss: 1.7206
- accuracy: 0.4677 - val_loss: 1.3296 - val_accuracy: 0.7530
Epoch 3/10
168/168 [==============================] - 5s 29ms/step - loss: 1.3213
- accuracy: 0.5981 - val_loss: 0.9415 - val_accuracy: 0.8555
Epoch 4/10
168/168 [==============================] - 5s 29ms/step - loss: 1.0495
- accuracy: 0.6854 - val_loss: 0.6883 - val_accuracy: 0.8689
Epoch 5/10
168/168 [==============================] - 5s 31ms/step - loss: 0.8421
- accuracy: 0.7372 - val_loss: 0.4917 - val_accuracy: 0.8750
Epoch 6/10
168/168 [==============================] - 5s 29ms/step - loss: 0.6806
- accuracy: 0.7974 - val_loss: 0.4055 - val_accuracy: 0.9123
Epoch 7/10
168/168 [==============================] - 5s 28ms/step - loss: 0.5870
- accuracy: 0.8178 - val_loss: 0.3217 - val_accuracy: 0.9379
Epoch 8/10
168/168 [==============================] - 5s 30ms/step - loss: 0.4839
- accuracy: 0.8532 - val_loss: 0.2688 - val_accuracy: 0.9410
Epoch 9/10
168/168 [==============================] - 5s 29ms/step - loss: 0.4278
- accuracy: 0.8735 - val_loss: 0.2196 - val_accuracy: 0.9661
Epoch 10/10
168/168 [==============================] - 5s 29ms/step - loss: 0.3656
- accuracy: 0.8868 - val_loss: 0.1770 - val_accuracy: 0.9657
```

Multi-outputs

Another common use-case for the **Functional API** is building models with multiple inputs or multiple outputs. Here, I'll show an example with one input and multiple outputs. **Note**, the example below is definitely fabricated, but I just want to demonstrate how to build a simple and small model. Most real-world examples are much bigger in terms of data and computing resources needed. You can find some nice examples here and here (albeit in Python, but you'll find that the Keras code and function names look almost the same as in R).

I'm using the car evaluation dataset from UC Irvine:

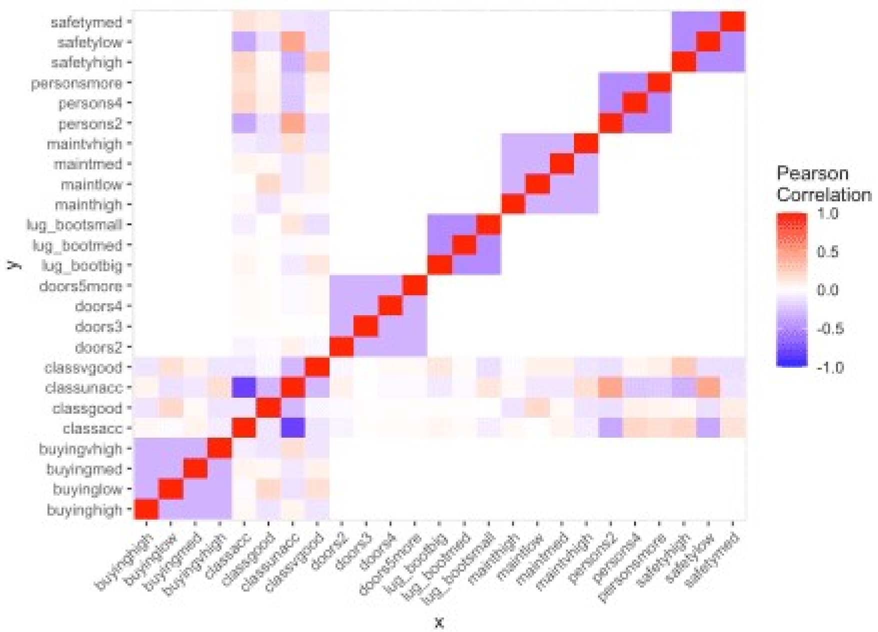
```
car_data <- readr::read_csv("car.data",
                col_names = c("buying", "maint", "doors", "persons",
"lug_boot", "safety", "class")) %>%
  remove_missing()
```

Categorical variables are converted into one-hot encoded dummy variables (in a real use-case, I'd probably not do that but use embeddings instead).

```
library(caret)
dmy <- dummyVars(" ~ doors + persons + buying + maint + lug_boot +
class + safety", data = car_data)
car_data <- data.frame(predict(dmy, newdata = car_data))
```

Just out of curiosity, I'm having a look at how my variables correlate:

```
cormat <- round(cor(car_data), 2)

cormat <- cormat %>%
  as_data_frame() %>%
  mutate(x = colnames(cormat)) %>%
  gather(key = "y", value = "value", doors2:safetymed)

cormat %>%
    remove_missing() %>%
    arrange(x, y) %>%
    ggplot(aes(x = x, y = y, fill = value)) +
    geom_tile() +
    scale_fill_gradient2(low = "blue", high = "red", mid = "white",
     midpoint = 0, limit = c(-1,1), space = "Lab",
     name = "Pearson\nCorrelation") +
    theme(axis.text.x = element_text(angle = 45, vjust = 1, hjust = 1))
```



Now, I'm preparing my two outputs: I want my model to predict

1. car class (as binary classification: unacceptable or not)
2. car safety (as multi-category classification: high, medium, low)

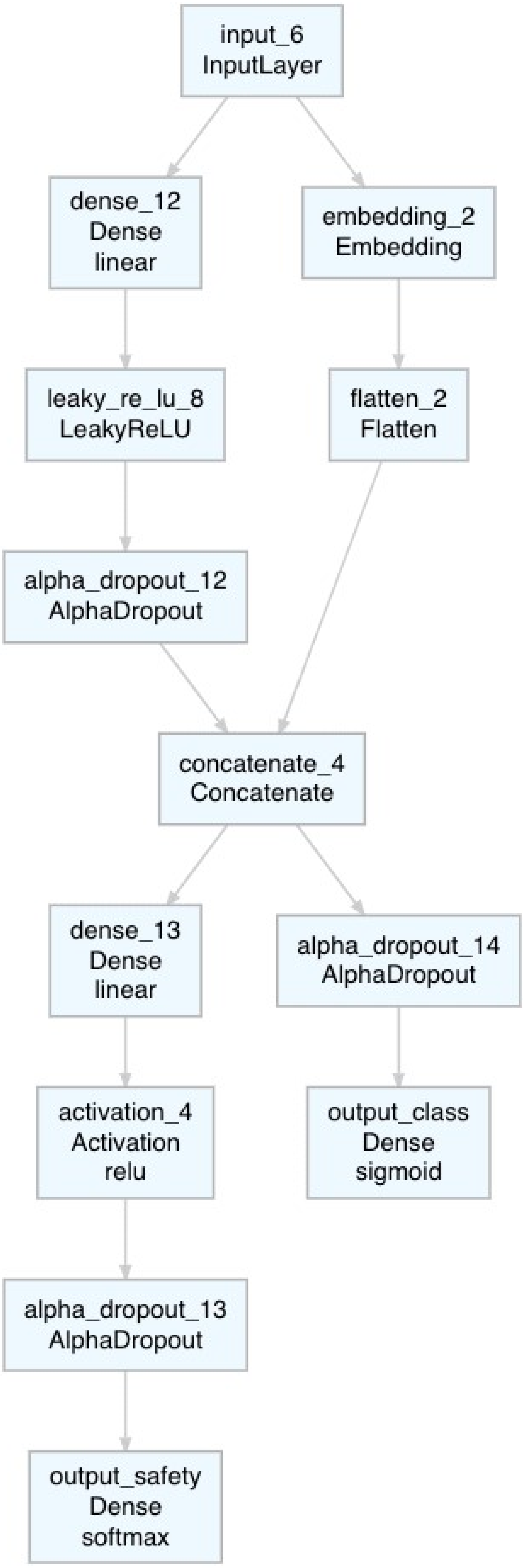
```
Y_log <- grepl("class|safety", colnames(car_data))
train_X <- as.matrix(car_data[, !Y_log])



Y_class <- as.matrix(car_data[, "classunacc"])
Y_safety <- as.matrix(car_data[, grepl("safety", colnames(car_data))])

train_Y_class <- Y_class[, ]
train_Y_safety <- Y_safety[, ]
```

Now, I'm defining my model architecture (this is not in any way optimized, nor is it a particularly useful architecture, but rather used to demonstrate a few options for using and combining layers):

- input layer just as before
- two parallel blocks: one embedding & one dense layer
- concatenating embedding & dense output
- output 1 (safety): dense layer + softmax for categorical crossentropy
- output 2 (class): sigmoid for binary crossentropy

```
# input layer
input_shape <- ncol(train_X)
input <- layer_input(shape = input_shape)

# embedding layer
embedding <- input %>%

  layer_embedding(input_dim = input_shape, output_dim = 36) %>%
  layer_flatten()

# dense layer
dense10 <- input %>%

  layer_dense(10) %>%
  layer_activation_leaky_relu() %>%
  layer_alpha_dropout(0.25)

# shared layer
shared <- layer_concatenate(c(embedding, dense10), axis = 1)

# output safety
output_n_safety <- ncol(Y_safety)

output_safety <- shared %>%

  layer_dense(5) %>%
  layer_activation("relu") %>%
  layer_alpha_dropout(0.25) %>%
```

```
  layer_dense(units = output_n_safety,
              activation = 'softmax',
              name = 'output_safety')

# output class
output_n_class <- ncol(Y_class)

output_class <- shared %>%

  layer_alpha_dropout(0.25) %>%

  layer_dense(units = output_n_class,
              activation = 'sigmoid', # with binary_crossentropy
              name = 'output_class')

# create model & compile
model <- keras_model(
  inputs = input,
  outputs = c(output_safety, output_class)
) %>%
  compile(
    loss = list(output_safety = "categorical_crossentropy",
                output_class = "binary_crossentropy"),
    optimizer = optimizer_adam(),
    metrics = "mse"
)



model %>% plot_model()
```


input_6
InputLayer
dense_12
Dense
linear
embedding_2
Embedding
leaky_re_lu_8
LeakyReLU
flatten_2
Flatten
alpha_dropout_12
AlphaDropout
concatenate_4
Concatenate
dense_13
Dense
linear
alpha_dropout_14
AlphaDropout
activation_4
Activation
relu
output_class
Dense
sigmoid
alpha_dropout_13
AlphaDropout
output_safety
Dense
softmax

```
model %>%
```

```
  fit(
    x = train_X,
    y = list(output_safety = train_Y_safety,
             output_class = train_Y_class),
    validation_split = 0.2,
    epochs = 25,
    batch_size = 32
)
```

```
Epoch 1/25
44/44 [==============================] - 1s 21ms/step - loss: 2.0614 -
output_safety_loss: 1.2896 - output_class_loss: 0.7718 -
output_safety_mse: 0.2585 - output_class_mse: 0.2803 - val_loss: 1.7722
- val_output_safety_loss: 1.0987 - val_output_class_loss: 0.6735 -
val_output_safety_mse: 0.2222 - val_output_class_mse: 0.2403
Epoch 2/25
44/44 [==============================] - 0s 6ms/step - loss: 1.9865 -
output_safety_loss: 1.2517 - output_class_loss: 0.7347 -
output_safety_mse: 0.2514 - output_class_mse: 0.2636 - val_loss: 1.7717
- val_output_safety_loss: 1.0988 - val_output_class_loss: 0.6729 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2404
Epoch 3/25
44/44 [==============================] - 0s 5ms/step - loss: 1.8930 -
output_safety_loss: 1.2187 - output_class_loss: 0.6744 -
output_safety_mse: 0.2452 - output_class_mse: 0.2385 - val_loss: 1.8024
- val_output_safety_loss: 1.0989 - val_output_class_loss: 0.7035 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2549
Epoch 4/25
44/44 [==============================] - 0s 5ms/step - loss: 1.8438 -
output_safety_loss: 1.2163 - output_class_loss: 0.6275 -
output_safety_mse: 0.2444 - output_class_mse: 0.2161 - val_loss: 1.8537
- val_output_safety_loss: 1.0990 - val_output_class_loss: 0.7548 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2758
Epoch 5/25
44/44 [==============================] - 0s 5ms/step - loss: 1.8193 -
output_safety_loss: 1.2125 - output_class_loss: 0.6068 -
output_safety_mse: 0.2450 - output_class_mse: 0.2073 - val_loss: 1.8755
- val_output_safety_loss: 1.0990 - val_output_class_loss: 0.7766 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2840
Epoch 6/25
44/44 [==============================] - 0s 5ms/step - loss: 1.7904 -
output_safety_loss: 1.1899 - output_class_loss: 0.6006 -
output_safety_mse: 0.2398 - output_class_mse: 0.2033 - val_loss: 1.8931
- val_output_safety_loss: 1.0991 - val_output_class_loss: 0.7941 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2902
Epoch 7/25
44/44 [==============================] - 0s 6ms/step - loss: 1.7696 -
output_safety_loss: 1.1811 - output_class_loss: 0.5885 -
output_safety_mse: 0.2372 - output_class_mse: 0.1989 - val_loss: 1.8554
- val_output_safety_loss: 1.0991 - val_output_class_loss: 0.7564 -
```

```
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2778
Epoch 8/25
44/44 [==============================] - 0s 5ms/step - loss: 1.7570 -
output_safety_loss: 1.1919 - output_class_loss: 0.5652 -
output_safety_mse: 0.2400 - output_class_mse: 0.1921 - val_loss: 1.8243
- val_output_safety_loss: 1.0992 - val_output_class_loss: 0.7251 -
val_output_safety_mse: 0.2224 - val_output_class_mse: 0.2670
Epoch 9/25
44/44 [==============================] - 0s 5ms/step - loss: 1.7234 -
output_safety_loss: 1.1578 - output_class_loss: 0.5655 -
output_safety_mse: 0.2339 - output_class_mse: 0.1912 - val_loss: 1.8038
- val_output_safety_loss: 1.0992 - val_output_class_loss: 0.7046 -
val_output_safety_mse: 0.2224 - val_output_class_mse: 0.2599
Epoch 10/25
44/44 [==============================] - 0s 5ms/step - loss: 1.7180 -
output_safety_loss: 1.1670 - output_class_loss: 0.5509 -
output_safety_mse: 0.2358 - output_class_mse: 0.1853 - val_loss: 1.7748
- val_output_safety_loss: 1.0993 - val_output_class_loss: 0.6755 -
val_output_safety_mse: 0.2224 - val_output_class_mse: 0.2492
Epoch 11/25
44/44 [==============================] - 0s 5ms/step - loss: 1.6670 -
output_safety_loss: 1.1498 - output_class_loss: 0.5173 -
output_safety_mse: 0.2321 - output_class_mse: 0.1736 - val_loss: 1.7404
- val_output_safety_loss: 1.0992 - val_output_class_loss: 0.6412 -
val_output_safety_mse: 0.2224 - val_output_class_mse: 0.2361
Epoch 12/25
44/44 [==============================] - 0s 5ms/step - loss: 1.6748 -
output_safety_loss: 1.1461 - output_class_loss: 0.5287 -
output_safety_mse: 0.2316 - output_class_mse: 0.1775 - val_loss: 1.7181
- val_output_safety_loss: 1.0992 - val_output_class_loss: 0.6189 -
val_output_safety_mse: 0.2224 - val_output_class_mse: 0.2272
Epoch 13/25
44/44 [==============================] - 0s 5ms/step - loss: 1.6274 -
output_safety_loss: 1.1239 - output_class_loss: 0.5035 -
output_safety_mse: 0.2269 - output_class_mse: 0.1690 - val_loss: 1.6934
- val_output_safety_loss: 1.0991 - val_output_class_loss: 0.5943 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2168
Epoch 14/25
44/44 [==============================] - 0s 5ms/step - loss: 1.6217 -
output_safety_loss: 1.1345 - output_class_loss: 0.4872 -
output_safety_mse: 0.2290 - output_class_mse: 0.1635 - val_loss: 1.6746
- val_output_safety_loss: 1.0991 - val_output_class_loss: 0.5755 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2092
Epoch 15/25
44/44 [==============================] - 0s 5ms/step - loss: 1.5843 -
output_safety_loss: 1.1255 - output_class_loss: 0.4588 -
output_safety_mse: 0.2280 - output_class_mse: 0.1523 - val_loss: 1.6546
- val_output_safety_loss: 1.0990 - val_output_class_loss: 0.5556 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.2011
Epoch 16/25
44/44 [==============================] - 0s 5ms/step - loss: 1.5974 -
output_safety_loss: 1.1255 - output_class_loss: 0.4720 -
```

```
output_safety_mse: 0.2280 - output_class_mse: 0.1589 - val_loss: 1.6377
- val_output_safety_loss: 1.0990 - val_output_class_loss: 0.5386 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.1941
Epoch 17/25
44/44 [==============================] - 0s 6ms/step - loss: 1.5751 -
output_safety_loss: 1.1243 - output_class_loss: 0.4508 -
output_safety_mse: 0.2275 - output_class_mse: 0.1506 - val_loss: 1.6203
- val_output_safety_loss: 1.0990 - val_output_class_loss: 0.5213 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.1867
Epoch 18/25
44/44 [==============================] - 0s 5ms/step - loss: 1.5572 -
output_safety_loss: 1.1044 - output_class_loss: 0.4528 -
output_safety_mse: 0.2233 - output_class_mse: 0.1515 - val_loss: 1.6252
- val_output_safety_loss: 1.0989 - val_output_class_loss: 0.5263 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.1891
Epoch 19/25
44/44 [==============================] - 0s 5ms/step - loss: 1.5833 -
output_safety_loss: 1.1199 - output_class_loss: 0.4634 -
output_safety_mse: 0.2268 - output_class_mse: 0.1564 - val_loss: 1.6150
- val_output_safety_loss: 1.0988 - val_output_class_loss: 0.5162 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.1847
Epoch 20/25
44/44 [==============================] - 0s 5ms/step - loss: 1.5536 -
output_safety_loss: 1.1153 - output_class_loss: 0.4383 -
output_safety_mse: 0.2256 - output_class_mse: 0.1476 - val_loss: 1.6119
- val_output_safety_loss: 1.0988 - val_output_class_loss: 0.5131 -
val_output_safety_mse: 0.2223 - val_output_class_mse: 0.1832
Epoch 21/25
44/44 [==============================] - 0s 5ms/step - loss: 1.5408 -
output_safety_loss: 1.1114 - output_class_loss: 0.4294 -
output_safety_mse: 0.2250 - output_class_mse: 0.1455 - val_loss: 1.6087
- val_output_safety_loss: 1.0987 - val_output_class_loss: 0.5100 -
val_output_safety_mse: 0.2222 - val_output_class_mse: 0.1814
Epoch 22/25
44/44 [==============================] - 0s 5ms/step - loss: 1.5481 -
output_safety_loss: 1.1143 - output_class_loss: 0.4339 -
output_safety_mse: 0.2256 - output_class_mse: 0.1452 - val_loss: 1.6057
- val_output_safety_loss: 1.0987 - val_output_class_loss: 0.5070 -
val_output_safety_mse: 0.2222 - val_output_class_mse: 0.1796
Epoch 23/25
44/44 [==============================] - 0s 7ms/step - loss: 1.5457 -
output_safety_loss: 1.1100 - output_class_loss: 0.4357 -
output_safety_mse: 0.2247 - output_class_mse: 0.1474 - val_loss: 1.5967
- val_output_safety_loss: 1.0987 - val_output_class_loss: 0.4980 -
val_output_safety_mse: 0.2222 - val_output_class_mse: 0.1745
Epoch 24/25
44/44 [==============================] - 0s 5ms/step - loss: 1.5310 -
output_safety_loss: 1.1046 - output_class_loss: 0.4264 -
output_safety_mse: 0.2235 - output_class_mse: 0.1434 - val_loss: 1.6002
- val_output_safety_loss: 1.0987 - val_output_class_loss: 0.5016 -
val_output_safety_mse: 0.2222 - val_output_class_mse: 0.1749
Epoch 25/25
```

```
44/44 [==============================] - 0s 5ms/step - loss: 1.5385 -
output_safety_loss: 1.1073 - output_class_loss: 0.4312 -
output_safety_mse: 0.2240 - output_class_mse: 0.1456 - val_loss: 1.6007
- val_output_safety_loss: 1.0987 - val_output_class_loss: 0.5020 -
val_output_safety_mse: 0.2222 - val_output_class_mse: 0.1751
```

```
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     2020-09-17
##
## ─ 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)
##  caret        * 6.0-86       2020-03-20 [1] CRAN (R 4.0.0)
##  cellranger     1.1.0        2016-07-27 [1] CRAN (R 4.0.0)
##  class          7.3-17       2020-04-26 [1] CRAN (R 4.0.2)
##  cli            2.0.2        2020-02-28 [1] CRAN (R 4.0.0)
##  codetools      0.2-16       2018-12-24 [1] CRAN (R 4.0.2)
##  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)
##  data.table     1.13.0       2020-07-24 [1] CRAN (R 4.0.2)
##  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)
##  deepviz      * 0.0.1.9000   2020-09-15 [1] Github
(andrie/deepviz@2a35de6)
##  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)
##  DiagrammeR     1.0.6.1      2020-05-08 [1] CRAN (R 4.0.2)
##  dials        * 0.0.8        2020-07-08 [1] CRAN (R 4.0.2)
##  DiceDesign     1.8-1        2019-07-31 [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)
## farver         2.0.3       2020-01-16 [1] CRAN (R 4.0.0)
## forcats      * 0.5.0       2020-03-01 [1] CRAN (R 4.0.0)
## foreach        1.5.0       2020-03-30 [1] CRAN (R 4.0.0)
## fs             1.5.0       2020-07-31 [1] CRAN (R 4.0.2)
## furrr          0.1.0       2018-05-16 [1] CRAN (R 4.0.2)
## future         1.18.0      2020-07-09 [1] CRAN (R 4.0.2)
## generics       0.0.2       2018-11-29 [1] CRAN (R 4.0.0)
## ggforce        0.3.2       2020-06-23 [1] CRAN (R 4.0.2)
## ggplot2      * 3.3.2       2020-06-19 [1] CRAN (R 4.0.2)
## ggraph         2.0.3       2020-05-20 [1] CRAN (R 4.0.2)
## ggrepel        0.8.2       2020-03-08 [1] CRAN (R 4.0.2)
## globals        0.12.5      2019-12-07 [1] CRAN (R 4.0.2)
## glue           1.4.2       2020-08-27 [1] CRAN (R 4.0.2)
## gower          0.2.2       2020-06-23 [1] CRAN (R 4.0.2)
## GPfit          1.0-8       2019-02-08 [1] CRAN (R 4.0.2)
## graphlayouts   0.7.0       2020-04-25 [1] CRAN (R 4.0.2)
## gridExtra      2.3         2017-09-09 [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)
## htmlwidgets    1.5.1       2019-10-08 [1] CRAN (R 4.0.0)
## httr           1.4.2       2020-07-20 [1] CRAN (R 4.0.2)
## igraph         1.2.5       2020-03-19 [1] CRAN (R 4.0.0)
## infer        * 0.5.3       2020-07-14 [1] CRAN (R 4.0.2)
## ipred          0.9-9       2019-04-28 [1] CRAN (R 4.0.0)
## iterators      1.0.12      2019-07-26 [1] CRAN (R 4.0.0)
## jsonlite       1.7.1       2020-09-07 [1] CRAN (R 4.0.2)
## keras        * 2.3.0.0.9000 2020-09-15 [1] Github
(rstudio/keras@ad737d1)
## knitr          1.29        2020-06-23 [1] CRAN (R 4.0.2)
## labeling       0.3         2014-08-23 [1] CRAN (R 4.0.0)
## lattice      * 0.20-41     2020-04-02 [1] CRAN (R 4.0.2)
## lava           1.6.7       2020-03-05 [1] CRAN (R 4.0.0)
## lhs            1.0.2       2020-04-13 [1] CRAN (R 4.0.2)
## lifecycle      0.2.0       2020-03-06 [1] CRAN (R 4.0.0)
## listenv        0.8.0       2019-12-05 [1] CRAN (R 4.0.2)
## 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)
## MASS           7.3-53      2020-09-09 [1] CRAN (R 4.0.2)
## 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)
## modeldata    * 0.0.2       2020-06-22 [1] CRAN (R 4.0.2)
## ModelMetrics   1.2.2.2     2020-03-17 [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)
## nlme           3.1-149     2020-08-23 [1] CRAN (R 4.0.2)
```

```
##  nnet            7.3-14       2020-04-26 [1] CRAN (R 4.0.2)
##  parsnip       * 0.1.3        2020-08-04 [1] CRAN (R 4.0.2)
##  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)
##  plyr            1.8.6        2020-03-03 [1] CRAN (R 4.0.0)
##  polyclip        1.10-0       2019-03-14 [1] CRAN (R 4.0.2)
##  prettyunits     1.1.1        2020-01-24 [1] CRAN (R 4.0.0)
##  pROC            1.16.2       2020-03-19 [1] CRAN (R 4.0.0)
##  processx        3.4.4        2020-09-03 [1] CRAN (R 4.0.2)
##  prodlim         2019.11.13   2019-11-17 [1] CRAN (R 4.0.0)
##  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)
##  RColorBrewer    1.1-2        2014-12-07 [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)
##  recipes       * 0.1.13       2020-06-23 [1] CRAN (R 4.0.2)
##  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)
##  reshape2        1.4.4        2020-04-09 [1] CRAN (R 4.0.0)
##  reticulate      1.16-9001    2020-09-15 [1] Github
(rstudio/reticulate@4f6a898)
##  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)
##  rpart           4.1-15       2019-04-12 [1] CRAN (R 4.0.2)
##  rprojroot       1.3-2        2018-01-03 [1] CRAN (R 4.0.0)
##  rsample       * 0.0.7        2020-06-04 [1] CRAN (R 4.0.2)
##  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)
##  survival        3.2-3        2020-06-13 [1] CRAN (R 4.0.2)
##  tensorflow    * 2.2.0.9000   2020-09-15 [1] Github
(rstudio/tensorflow@bb4062a)
##  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)
##  tidygraph       1.2.0        2020-05-12 [1] CRAN (R 4.0.2)
##  tidymodels    * 0.1.1        2020-07-14 [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)
##  timeDate        3043.102     2018-02-21 [1] CRAN (R 4.0.0)
##  tune          * 0.1.1        2020-07-08 [1] CRAN (R 4.0.2)
##  tweenr          1.0.1        2018-12-14 [1] CRAN (R 4.0.2)
##  usethis         1.6.1        2020-04-29 [1] CRAN (R 4.0.0)
##  vctrs           0.3.4        2020-08-29 [1] CRAN (R 4.0.2)
```

```
##  viridis        0.5.1       2018-03-29 [1] CRAN (R 4.0.2)
##  viridisLite    0.3.0       2018-02-01 [1] CRAN (R 4.0.0)
##  visNetwork     2.0.9       2019-12-06 [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)
##  workflows    * 0.1.3       2020-08-10 [1] CRAN (R 4.0.2)
##  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)
##  yardstick    * 0.0.7       2020-07-13 [1] CRAN (R 4.0.2)
##  zeallot        0.1.0       2018-01-28 [1] CRAN (R 4.0.0)
##
## [1] /Library/Frameworks/R.framework/Versions/4.0/Resources/library
```