R/Keras Examples for FFNs & CNNs

PUBH 8475/Stat 8056

Based on “Deep Learning with R”
https://www.manning.com/books/deep-learning-with-r
(including source code)
install.packages("keras")

#Install the core Keras library and TensorFlow:

library(keras)

#CPU-based:
install_keras()

#GPU-based:

install_keras(tensorflow = "gpu")

#Remark: I failed first time when I tried to install on my old laptop with an older version of R; after I updated the R to the current version, it worked!
**Alternative installation**: R Interface to TF (and Keras)

Step 1. Install TF: Go to

[https://tensorflow.rstudio.com/](https://tensorflow.rstudio.com/)

click on Installation:

[https://tensorflow.rstudio.com/installation/](https://tensorflow.rstudio.com/installation/)

Installation
First, install the tensorflow R package from GitHub as follows:

```
install.packages("tensorflow")
```

Then, use the `install_tensorflow()` function to install TensorFlow. Note that on Windows you need a working installation of Anaconda.

```
library(tensorflow)
install_tensorflow()
```

You can confirm that the installation succeeded with:

```
library(tensorflow)
tf$constant("Hellow Tensorflow")
```

## tf.Tensor(b'Hellow Tensorflow', shape=(), dtype=string)

This will provide you with a default installation of TensorFlow suitable for use with the tensorflow R package.

Step 2. Install Keras: in R,

```
> install.packages("keras")
```
MNIST dataset
#Loading the MNIST dataset in Keras:
library(keras)
mnist <- dataset_mnist()
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnistitest$x
test_labels <- mnist$test$y

str(train_images)
# int [1:60000, 1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ... 
str(test_labels)
# int [1:10000(1d)] 7 2 1 0 4 1 4 9 5 9 ...

train_images <- array_reshape(train_images, c(60000, 28 * 28))
train_images <- train_images / 255

test_images <- array_reshape(test_images, c(10000, 28 * 28))
test_images <- test_images / 255

train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)
```r
network <- keras_model_sequential() %>%
  layer_dense(units = 512, activation = "relu", input_shape = c(28 * 28)) %>%
  layer_dense(units = 10, activation = "softmax")

network %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

network %>% fit(train_images, train_labels, epochs = 5, batch_size = 128)
#Epoch 1/5  469/469 [==============================] - 2s 4ms/step - loss: 0.2575 - accuracy: 0.9254
#Epoch 2/5  469/469 [==============================] - 2s 3ms/step - loss: 0.1031 - accuracy: 0.9695
#Epoch 3/5  469/469 [==============================] - 2s 3ms/step - loss: 0.0688 - accuracy: 0.9792
#Epoch 4/5  469/469 [==============================] - 2s 3ms/step - loss: 0.0489 - accuracy: 0.9857
#Epoch 5/5  469/469 [==============================] - 2s 3ms/step - loss: 0.0380 - accuracy: 0.9887

metrics <- network %>% evaluate(test_images, test_labels, verbose = 0)
metrics
#  loss   accuracy
# 0.06546536 0.98079997
```
Model
Model: "sequential_2"

Layer (type)                  Output Shape         Param #
==================================================================================
dense_5 (Dense)              (None, 512)          401920

dense_4 (Dense)              (None, 10)           5130

Total params: 407,050
Trainable params: 407,050
Non-trainable params: 0
Setting aside a validation set:
val_indices <- 1:10000
x_val <- train_images[val_indices,]
partial_x_train <- train_images[-val_indices,]
y_val <- train_labels[val_indices,]
partial_y_train <- train_labels[-val_indices,]

history <- network %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)

plot(history)
> str(history)
List of 2
  $ params :List of 3
    ..$ verbose: int 1
    ..$ epochs : int 20
    ..$ steps : int 98
  $ metrics:List of 4
    ..$ loss        : num [1:20] 0.00104 0.000786 0.000423 0.000472 0.000307 ... 
    ..$ accuracy    : num [1:20] 1 1 1 1 1 ...
    ..$ val_loss : num [1:20] 0.00126 0.00127 0.00111 0.00124 0.00136 ... 
    ..$ val_accuracy: num [1:20] 1 1 1 1 1 ...
- attr(*, "class")= chr "keras_training_history"
```r
network <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu", input_shape = c(28 * 28)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")

network %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

network %>% fit(train_images, train_labels, epochs = 5, batch_size = 128)

metrics <- network %>% evaluate(test_images, test_labels, verbose = 0)
metrics
# loss  accuracy
#0.1524482 0.9538000
```
```
> network
Model
Model: "sequential_3"

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense_8 (Dense)</td>
<td>(None, 32)</td>
<td>25120</td>
</tr>
<tr>
<td>dense_7 (Dense)</td>
<td>(None, 16)</td>
<td>528</td>
</tr>
<tr>
<td>dense_6 (Dense)</td>
<td>(None, 10)</td>
<td>170</td>
</tr>
</tbody>
</table>

Total params: 25,818
Trainable params: 25,818
Non-trainable params: 0
```
library(keras)

model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu", input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")

model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)

model %>% compile(
  optimizer = optimizer_rmsprop(lr=0.001),
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
# Loading the IMDB dataset:
library(keras)

imdb <- dataset_imdb(num_words = 10000)
c(c(train_data, train_labels), c(test_data, test_labels)) %<-% imdb
## the above is equivalent to:
imdb <- dataset_imdb(num_words = 10000)
train_data <- imdb$train$x
train_labels <- imdb$train$y
test_data <- imdb$test$x
test_labels <- imdb$test$y

# .......

# Setting aside a validation set:
val_indices <- 1:10000
x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]
y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
#training your model:

```r
model %>% compile(
  optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = c("accuracy")
)
```

```r
history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  batch_size = 512,
  validation_data = list(x_val, y_val)
)
```
> str(history)

List of 2

$ params: List of 8
  ..$ metrics : chr [1:4] "loss" "acc" "val_loss" "val_acc"
  ..$ epochs : int 20
  ..$ steps : NULL
  ..$ do_validation : logi TRUE
  ..$ samples : int 15000
  ..$ batch_size : int 512
  ..$ verbose : int 1
  ..$ validation_samples: int 10000

$ metrics: List of 4
  ..$ acc : num [1:20] 0.783 0.896 0.925 0.941 0.952 ...
  ..$ loss : num [1:20] 0.532 0.331 0.24 0.186 0.153 ...
  ..$ val_acc: num [1:20] 0.832 0.882 0.886 0.888 0.888 ...
  ..$ val_loss: num [1:20] 0.432 0.323 0.292 0.278 0.278 ...

attr(*, "class")= chr "keras_training_history"
library(keras)

model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3), activation = "relu",
                input_shape = c(28, 28, 1)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3), activation = "relu") %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")
```plaintext
> model

<table>
<thead>
<tr>
<th>Layer (type)</th>
<th>Output Shape</th>
<th>Param #</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv2d_1 (Conv2D)</td>
<td>(None, 26, 26, 32)</td>
<td>320</td>
</tr>
<tr>
<td>maxpooling2d_1 (MaxPooling2D)</td>
<td>(None, 13, 13, 32)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d_2 (Conv2D)</td>
<td>(None, 11, 11, 64)</td>
<td>18496</td>
</tr>
<tr>
<td>maxpooling2d_2 (MaxPooling2D)</td>
<td>(None, 5, 5, 64)</td>
<td>0</td>
</tr>
<tr>
<td>conv2d_3 (Conv2D)</td>
<td>(None, 3, 3, 64)</td>
<td>36928</td>
</tr>
<tr>
<td>flatten_1 (Flatten)</td>
<td>(None, 576)</td>
<td>0</td>
</tr>
<tr>
<td>dense_1 (Dense)</td>
<td>(None, 64)</td>
<td>36928</td>
</tr>
<tr>
<td>dense_2 (Dense)</td>
<td>(None, 10)</td>
<td>650</td>
</tr>
</tbody>
</table>

Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0
```
mnist <- dataset_mnist()
c(c(train_images, train_labels), c(test_images, test_labels)) %<-% mnist
train_images <- array_reshape(train_images, c(60000, 28, 28, 1))
train_images <- train_images / 255
test_images <- array_reshape(test_images, c(10000, 28, 28, 1))
test_images <- test_images / 255
train_labels <- to_categorical(train_labels)
test_labels <- to_categorical(test_labels)
model %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
model %>% fit(
  train_images, train_labels,
  epochs = 5, batch_size=64
)

> results <- model %>% evaluate(test_images, test_labels)
> results
$loss
[1] 0.02563557
$acc
[1] 0.993
Visualizing what CNNs learn

1. Visualizing intermediate activations:
   Input image; 2\textsuperscript{nd} and 7\textsuperscript{th} feature maps in the first layer
List out every feature map in every layer
2. Visualizing CNN filters:

• Input maximizing the response of a filter: top to bottom layers in VGG for ImageNet
3. Visualizing class activation maps (CAM): Grad-CAM

Still an active research topic; see Dai et al (2022) using R2. More generally, explainable AI (XAI) (especially for black-box methods).