TensorFlow with R

Eric Smrkovsky

11. 12. 2022

1 Introduction

The computer's ability to process and analyze data has been developing for about 70 years. Today, we recognize Machine Learning (ML) as an essential sub-field of Artificial Intelligence (AI), allowing computers to make decisions based on the data given as input to ML algorithms. Supervised ML typically creates a function from labeled training data to be used for some validation task with unlabeled data. Many scientists and engineers use supervised learning for practical tasks, scientific studies, and data analysis. Some examples include: analyzing the brain's electrical activity through signal processing, speech recognition, natural language processing, and computer vision(Das et al. 2015).

Moving farther down the rabbit hole of AI, deep learning is a sub-field of machine learning that uses artificial neural networks. Typically scientists think of a neural network as a collection of nodes that models the neurons in the brain. This report introduces the R package version of one of the most popular AI Python libraries, TensorFlow. This report introduces the TensorFlow package's functionality and then presents a tutorial on implementing a simple neural network using the MNIST database as input.

This document is designed to be reproducible so that the reader can replicate and execute the code snippets within a local RStudio environment. The source code for this document can serve as a template for creating RMarkdown reports and can be found here: https://github.com/Ericsmrk/Math-105-Coursework. This Github repository link also has the .html and .rmd versions of this document.

2 TensorFlow R Package

TensorFlow is a deep learning library created by Google offering high and low-level capabilities for deploying deep learning models. Keras is used to define neural networks and runs on top of TensorFlow. The easy programming interface of Keras is used to build the model then TensorFlow does all the work executing the model. TensorFlow uses multidimensional arrays (Tensors) for all mathematical operations and machine learning operations.

2.1 Installation of TensorFlow and Keras package

You will need to have R, RStudio and Python installed on your computer.

For R and RStudio, the installation is found here: https://posit.co/download/rstudio-desktop/

For Python the latest version can be found here: https://www.python.org/downloads/

Running the following code will set you up to start working with TensorFlow in RStudio.

```
install.packages("keras")
library(keras)
install_keras(envname = "r-reticulate")
library(tensorflow)
```

You can check to see if you were successful with your installation tasks by running the following code chunk.

```
tf$constant("Hello Tensorflow!")
```

2.2 Tensors, Variables, and Graphs

2.2.1 Tensors

A "Tensor" in TensorFlow is an immutable (can not be changed) object with a uniform type. Here are some examples of the different data types of Tensors.

```
names(tf$dtypes)
```

```
## Loaded Tensorflow version 2.9.3
```

```
##  [1] "as_dtype"         "bfloat16"         "bool"             "cast"
##  [5] "complex"          "complex128"       "complex64"        "double"
##  [9] "DType"            "float16"          "float32"          "float64"
## [13] "half"             "int16"            "int32"            "int64"
## [17] "int8"             "qint16"           "qint32"           "qint8"
## [21] "QUANTIZED_DTYPES" "quint16"          "quint8"           "resource"
## [25] "saturate_cast"    "string"           "uint16"           "uint32"
## [29] "uint64"           "uint8"            "variant"
```

The structure of a Tensor is described as it's shape and has the following features:
* Dimension
* Rank

* Axis
* Size

A Tensor's rank describes the number of dimensions of the Tensor. The shape of the Tensor is the number of elements in each dimension. The axis is the particular dimension. The size is the total number of elements within the Tensor.

Here is an example of a scalar, a vector, and a 2 dimensional matrix.

```
aScalar <- as_tensor(32, dtype = "int32")
aVector <- as_tensor(c(45:50), dtype = "int32")
aMatrix2 <- as_tensor(rbind(c(51, 52),
                            c(53, 54),
                            c(55, 56)), dtype = "float64")
print(aScalar)
```

```
## tf.Tensor(32, shape=(), dtype=int32)
```

```
print(aVector)
```

```
## tf.Tensor([45 46 47 48 49 50], shape=(6), dtype=int32)
```

```
print(aMatrix2)
```

```
## tf.Tensor(
## [[51. 52.]
##  [53. 54.]
##  [55. 56.]], shape=(3, 2), dtype=float64)
```

Tensors can be used for a wide range of mathematical operations. For example, we can divide a vector by a scalar.

```
aDevProblem <- tf$math$divide(aVector,aScalar)
print(aDevProblem)
```

```
## tf.Tensor([1.40625 1.4375  1.46875 1.5     1.53125 1.5625 ], shape=(6), dtype=float64
```

After the computation we convert to an array to view the results without the shape properties being shown.

```
as.array(aDevProblem)
```

```
## [1] 1.40625 1.43750 1.46875 1.50000 1.53125 1.56250
```

There are plenty of other mathematical operations that can be done with Tensors including some that are very specific to deep learning tasks. For example, in our neural network example below the final layer of the model uses a softmax layer which is a complex computation that is simplified by TensorFlow.

```
tf$nn$softmax(aMatrix2)
```

```
## tf.Tensor(
## [[0.26894142 0.73105858]
##  [0.26894142 0.73105858]
##  [0.26894142 0.73105858]], shape=(3, 2), dtype=float64)
```

2.2.2 Variables

A variable in TensorFlow is a data structure that has a Tensor at it's core and acts just like a Tensor. A variable has it's own shape, dtype, etc. Variables are used to share and access a persistent state within the program.

```
aMatrix2_variable <- tf$Variable(aMatrix2)
```

Most mathematical operations work on variables as well.

```
tf$nn$softmax(aMatrix2_variable)
```

```
## tf.Tensor(
## [[0.26894142 0.73105858]
##  [0.26894142 0.73105858]
##  [0.26894142 0.73105858]], shape=(3, 2), dtype=float64)
```

2.2.3 Graphs

TensorFlow graphs can encapsulate R functions and export them for use elsewhere. These graphs can be executed without using R and within other programs, on a website, or even within a mobile app. Using 'tf_function()' is very similar to using the R 'function()' except that 'function()' is eagerly executed and 'tf_function()' uses graph execution. Graph execution allows TensorFlow to run more efficiently by only executing lines of code once when the line is not needed to be run dynamically. Here is an example of a standard R 'function().'

```r
# R function
aFunction <- function(x,y) {
  message('Do Division')
  tf$math$divide(x,y)
}
```

Here is an example of using a 'tf_function()' as a wrapper for our newly created 'aFunction'.

```r
# TensorFlow function
graphFunction <- tf_function(aFunction)
```

We create two Tensors and test these functions.

```r
xTest <- as_tensor(1:2, "int32", shape = c(1, 2))
yTest <- as_tensor(2:3, "int32", shape = c(2, 1))
aFunction(xTest,yTest)
```

```
## Do Division

## tf.Tensor(
## [[0.5        1.        ]
##  [0.33333333 0.66666667]], shape=(2, 2), dtype=float64)
```

```r
graphFunction(xTest,yTest)
```

```
## Do Division

## tf.Tensor(
## [[0.5        1.        ]
##  [0.33333333 0.66666667]], shape=(2, 2), dtype=float64)
```

Noticeably, the results from these functions are the same, which is expected. So what makes them different? The 'message' statement is code that can only be executed once, as that line of code is not needed for the function to perform its operation. Let us execute the functions again.

```r
aFunction(xTest,yTest)
```

```
## Do Division

## tf.Tensor(
## [[0.5        1.        ]
##  [0.33333333 0.66666667]], shape=(2, 2), dtype=float64)
```

```
graphFunction(xTest,yTest)
```

```
## tf.Tensor(
## [[0.5        1.        ]
##  [0.33333333 0.66666667]], shape=(2, 2), dtype=float64)
```

We find that the 'message' statement only runs the first time for the graph execution function, while it runs twice during the eager execution function. Imagine if there were 50 lines of code generating a neural network that could be executed once and encapsulated into a Tensor graph saving a data scientist countless hours running the model. That is the POWER of TensorFlow.

The above simple division function generates the following graph structure below. Please review this hierarchy and see if you can understand the graph structure, its attributes, and what hard-coded operations are accessed.

```
graphFunction$get_concrete_function(as_tensor(10),as_tensor(2))$graph$as_graph_def()
```

```
## Do Division
```

```
## node {
##   name: "x"
##   op: "Placeholder"
##   attr {
##     key: "_user_specified_name"
##     value {
##       s: "x"
##     }
##   }
##   attr {
##     key: "dtype"
##     value {
##       type: DT_FLOAT
##     }
##   }
##   attr {
##     key: "shape"
##     value {
##       shape {
##       }
##     }
##   }
## }
## node {
##   name: "y"
##   op: "Placeholder"
##   attr {
##     key: "_user_specified_name"
##     value {
##       s: "y"
##     }
##   }
##   attr {
##     key: "dtype"
##     value {
##       type: DT_FLOAT
##     }
##   }
##   attr {
##     key: "shape"
##     value {
##       shape {
##       }
##     }
##   }
## }
## node {
##   name: "truediv"
##   op: "RealDiv"
##   input: "x"
##   input: "y"
##   attr {
##     key: "T"
##     value {
##       type: DT_FLOAT
##     }
##   }
## }
## node {
##   name: "Identity"
##   op: "Identity"
##   input: "truediv"
##   attr {
##     key: "T"
##     value {
##       type: DT_FLOAT
##     }
##   }
## }
## versions {
##   producer: 1087
## }
```

3 Neural Network using Keras

Keras is an Application Programming Interface used for building deep learning models and runs on-top of TensorFlow. Keras is known for being simple, flexible, and powerful. Keras provides a way for data scientists to go from an idea to results quickly. The idea behind Keras is to build a model by stacking layers. The sequential model creates a linear stack of layers, and the functional API creates a graph of many layers. While the Keras functional API is beyond the scope of this paper, we present an example of the sequential model below.

3.1 Keras Sequential Model Example

First, we load the MNIST database directly from the Keras package. This provides us with lists that include training and validation datasets that we will use to train the sequential model. Notice how we convert the example data from integers to floating-point numbers and normalize the pixel range. This changes the data points to show 0 to 1 rather than 0 to 255.

```r
db <- dataset_mnist()
c(c(x_train, y_train), c(x_test, y_test)) %<-% db
x_train <- x_train / 255
x_test <-  x_test / 255
```

Now we have our data ready to go. We will use Keras to build a simple neural network. In the following code chunk, we create the layers in a “stack” for the sequential model. We define the input size to represent the two dimensional tensor filled with pixels.

```r
model <- keras_model_sequential(input_shape = c(28, 28)) %>%
  layer_flatten() %>%
  layer_dense(128, activation = "relu") %>%
  layer_dropout(0.2) %>%
  layer_dense(10)
model
```

```
## Model: "sequential"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                      Param #
## ================================================================================
##  flatten (Flatten)                  (None, 784)                       0
##  dense_1 (Dense)                     (None, 128)                     100480
##  dropout (Dropout)                   (None, 128)                     0
##  dense (Dense)                       (None, 10)                      1290
## ================================================================================
## Total params: 101,770
## Trainable params: 101,770
## Non-trainable params: 0
## ________________________________________________________________________________
```

As you can see above we have created a model with some layers. First layer flattens the input. Second layer adds a densely connected layer of size 28 by 28 and utilizes an activation function. Third layer provides dropout for the input preventing overfitting. Finally we combine some results with the final softmax layer.

Next we need to make some predictions to evaluate the model with. The following code chunk creates the probabilities for each class and stores them in a Tensor.

```r
predictions <- predict(model, x_train[1:2, , ])
tf$nn$softmax(predictions)
```

```
## tf.Tensor(
## [[0.05020111 0.165933   0.0560842  0.19874796 0.0505665  0.09155925
##   0.1289973  0.17099466 0.02586247 0.06105355]
##  [0.05179446 0.19439009 0.05563388 0.14096179 0.03854122 0.111488
##   0.08515134 0.15886442 0.05183514 0.11133965]], shape=(2, 10), dtype=float64)
```

The following code chunk creates a loss function and sets the hyper parameters for the network. The loss function used takes our probabilities and the ground truth associated with the examples and returns the loss. Next we set the optimizer to adam and set the metrics parameter to accuracy.

```r
loss_fn <- loss_sparse_categorical_crossentropy(from_logits = TRUE)
loss_fn(y_train[1:2], predictions)
```

```
## tf.Tensor(2.675620515062917, shape=(), dtype=float64)
```

```r
model %>% compile(
  optimizer = "adam",
  loss = loss_fn,
  metrics = "accuracy"
)
```

Finally we can fit the model to the data and determine the accuracy of our neural network.

```
model %>% fit(x_train, y_train, epochs = 5)
model %>% evaluate(x_test,  y_test, verbose = 2)
```

```
##       loss   accuracy
## 0.07641823 0.97710001
```

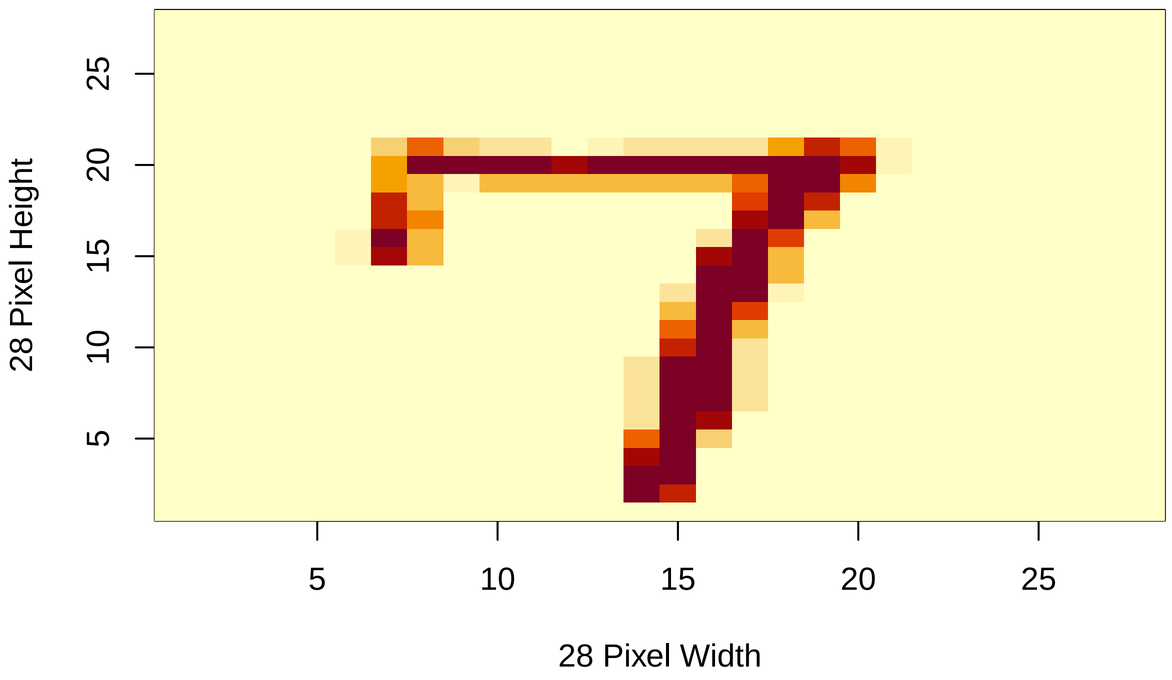
From these lines of R code you can see that the neural network created above performs well with classifying MNIST digits!

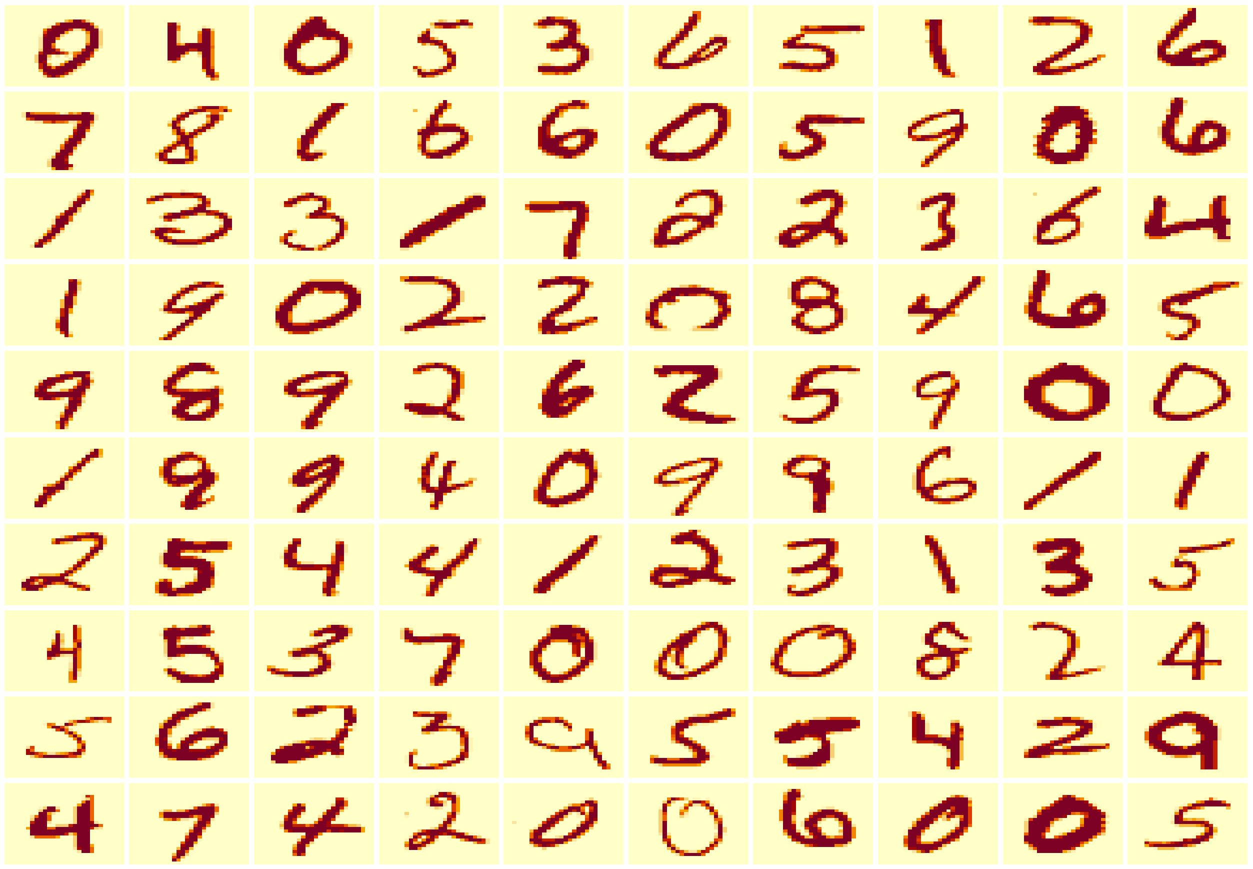
4 The MNIST Database

The data used for training and classification is the famous Modified National Institute of Standards and Technology (MNIST) database. The MNIST database was created in 1994 and released to the public in 1998 with the copyright held by Dr. Yann LeCun and Dr. Corinna Cortes(LeCun, Cortes, and Burges 2010). The MNIST database was constructed during a study by LeCun and colleagues regarding document recognition using various machine learning techniques(Lecun et al. 1998). The database eventually became the standard test for pattern recognition algorithms. The database consists of 60,000 training and 10,000 test images, containing 28 rows and 28 columns of black-and-white pixels that form handwritten digits. A review where sixty-eight machine learning classifiers were evaluated with the database found that neural net classifiers perform significantly better than other classifiers(Deng 2012). Below you see an example of some pixel data followed by images containing accurate representations of MNIST digits. Code for this can be found in the appendix. This code uses a smaller MNIST database found here: https://www.kaggle.com/competitions/digit-recognizer/data?select=train.csv

label	pixel122	pixel123	pixel124	pixel125	pixel126	pixel127
1	0	0	0	0	0	0
0	18	30	137	137	192	86
1	0	0	3	141	139	3
4	0	0	0	0	0	0
0	25	130	155	254	254	254
0	0	0	3	141	202	254
7	0	0	0	0	0	0
3	0	0	0	0	0	0
5	0	0	0	0	0	0
3	136	147	254	255	199	111

Lucky 7: 784 Pixels





5 Conclusion

This RMarkdown document has presented an introduction to the functionality of TensorFlow and provides an example of using Keras to build a neural network. Using these powerful Python packages within R can provide advantages over other packages by using the graph features of TensorFlow and the many different models accessible from Keras. Using the MNIST database, a data scientist can practice creating models to classify the digits using these packages. The reader of this report can find more resources for learning about these packages at the websites below.

https://tensorflow.rstudio.com/
https://www.tensorflow.org/
https://keras.io/

6 Appendix

6.1 Code to Display Mnist Digits

```r
## Load the training data
setwd("G:/My Drive/105/MNIST_Project")# choose folder where data is stored
train <- read.csv("data/train.csv")   # retrieve training data from local
cols <- c(1,124:129)                  # columns to view
kable(train[1:10,cols])               # show 10 rows of columns 1, 124-129
```

```r
## View an example: number 7
seven <- matrix(unlist(train[7,-1]),nrow = 28,byrow = T)
im <- t(apply(seven, 2, rev))
image(x = 1:28,
      y = 1:28,
      z = im,
      main = "Lucky 7: 784 Pixels",
      xlab = "28 Pixel Width",
      ylab = "28 Pixel Height")
```

```r
## Algorithm to View a 10 by 10 matrix filled with examples
set.seed(15)
data <- sample(as.integer(row.names(train)),100)# get 100 image examples
par(mfrow=c(10,10),mar=c(0.1,0.1,0.1,0.1)) # multiple graphs in a single plot

for (imageSample in data){  # for each image collect it's pixels and display
  row <- NULL
  for (pixel in 2:785)      # each image has 784 pixels
    row[pixel-1] <- train[imageSample,pixel] # store pixels for current example

  matrix1 <- matrix(row,28,28,byrow=FALSE)   # stores pixels backwards
  matrix2 <- matrix(rep(0,784),28,28)        # blank matrix to reorder pixels

  for (i in 1:28)
    for (j in 1:28)                          # reorder the columns into matrix2
      matrix2[i,28-j+1] <- matrix1[i,j]

  image(matrix2, axes=FALSE)                 # show numbers
}
rm(i,pixel,j,imageSample,row,matrix1,matrix2, cols, im, seven) # cleanup
```

7 Bibliography

Das, Sumit, Aritra Dey, Akash Pal, and Nabamita Roy. 2015. "Applications of Artificial Intelligence in Machine Learning: Review and Prospect." *International Journal of Computer Applications* 115 (9).

Deng, Li. 2012. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]." *IEEE Signal Processing Magazine* 29 (6): 141–42. https://doi.org/10.1109/MSP.2012.2211477.

LeCun, Yann, Corinna Cortes, and CJ Burges. 2010. "MNIST Handwritten Digit Database." *ATT Labs [Online]. Available: Http://Yann.lecun.com/Exdb/Mnist* 2.

Lecun, Y, L Bottou, Y Bengio, and P Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–2324.