Big Data Analytics

Lecture 9:

Applied Econometrics with Spark; Machine Learning and GPUs

Prof. Dr. Ulrich Matter

06/05/2021

Updates

Schedule

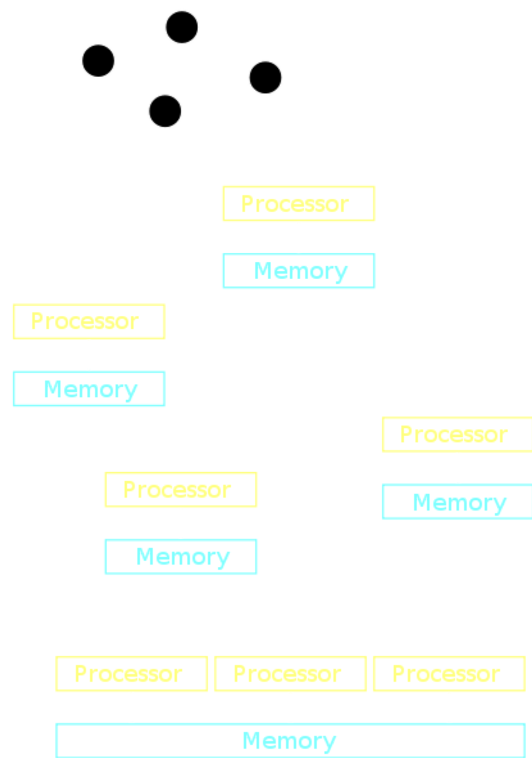
1. Introduction: Big Data, Data Economy. Walkowiak (2016): Chapter 1.
2. Computation and Memory in Applied Econometrics.
3. Computation and Memory in Applied Econometrics II.
4. Advanced R Programming. Wickham (2019): Chapters 2, 3, 17,23, 24.
5. Import, Cleaning and Transformation of Big Data. Walkowiak (2016): Chapter 3: p. 74-118.
6. Aggregation and Visualization. Walkowiak (2016): Chapter 3: p. 118-127; Wickham et al.(2015); Schwabish (2014).
7. Data Storage, Databases Interaction with R. Walkowiak (2016): Chapter 5.
8. Cloud Computing: Introduction/Overview, Distributed Systems, Walkowiak (2016): Chapter 4.
9. **Applied Econometrics with Spark; Machine Learning and GPUs.**
10. **Announcement Take-Home Exercises, Q&A in Zoom.**
11. **Feedback, Q&A in Zoom, (hand in presentations).**

Feedback

- Course Evaluation
- Additional course evaluation: Online Teaching Spring 2020
- Both open on Canvas/StudyNet!
- Please report technical problems!

Distributed Systems/MapReduce

Distributed systems



(a), (b): a distributed system. (c): a parallel system.I llustration by Miym. CC BY-SA 3.0

Map/Reduce with Hadoop

Map/Reduce with Hadoop

- Basic Hadoop installation comes with a few examples for very typical map/reduce programs.
- More sophisticated programs need to be custom made, written in Java.

Map/Reduce with Hadoop

```
# run mapreduce word count
# the specific hadoop program with the implemented wordcount is stored in a java-archive application
/usr/local/hadoop/bin/hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2
```

Map/Reduce with Hadoop

```
cat ~/wc_example/*
```

```
## Ann  2
## Becky    2
## Simon    2
## a    3
## friend   3
## is   3
## not  1
## of   3
```

MapReduce/Hadoop summary

- Motivation: need to scale out storage/memory.
- Allows for massive horizontal scaling.
- RAM/storage distributed across machines.

Applied Econometrics with Apache Spark

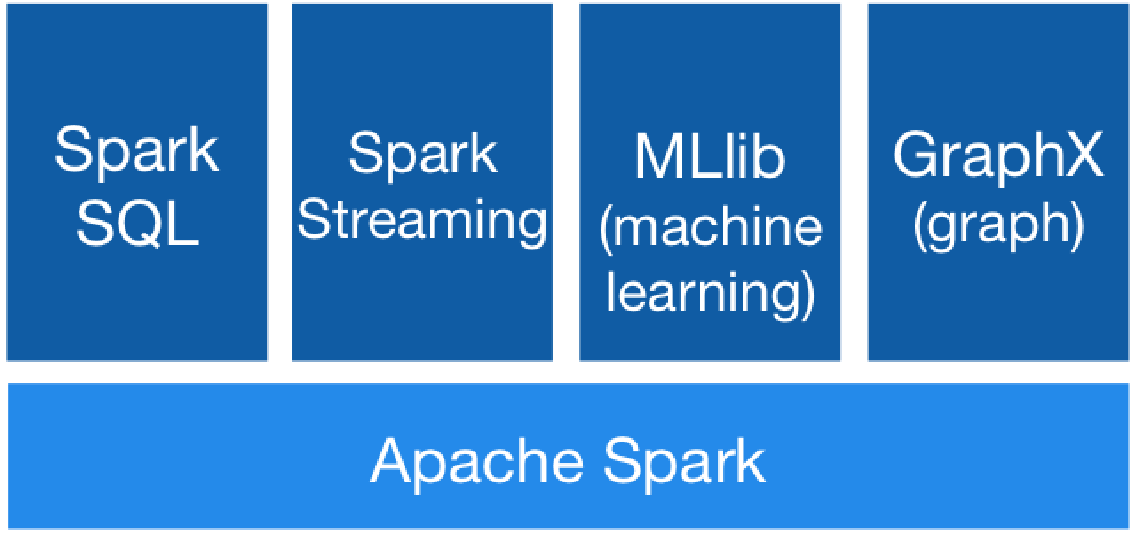
APACHE

Spark™

Spark basics

- Cluster computing platform made for data analytics.
- Based on Hadoop, but much faster in many core data analytics tasks.
- Easy to use from R.

Spark basics



Basic Spark stack (source: https://spark.apache.org/images/spark-stack.png)

Spark basics: RDDs

- Fundamental data structure: **resilient distributed dataset' (RDD)**.
- Distributed collections of elements.
- Manipulations are executed in parallel in these RDDs.

Spark in R

- Two prominent packages connect R to Spark: `SparkR` and RStudio's `sparklyr`.
- Similarly easy to use and cover all the basics for analytics tasks.
- See, e.g., this blog post for pros and cons.

SparkR in action

Note: by default, Spark/SparkR runs on a local standalone session (no cluster computer needed to learn Spark, test code etc.!)

```
# install.packages("SparkR")

# load packages
library(SparkR)

# start session
sparkR.session()
```

```
## Launching java with spark-submit command /home/umatter/.cache/spark/spark-2.4.5-bin-hadoop2.7/bin/s
```

```
## Java ref type org.apache.spark.sql.SparkSession id 1
```

Data import and summary statistics

We analyze the already familiar `flights.csv` dataset.

```
# Import data and create a SparkDataFrame (a distributed collection of data, RDD)
flights <- read.df(path="../data/flights.csv", source = "csv", header="true")
```

Data import and summary statistics

- ‘Import’ means creating the RDDs and an R-object pointing to the data.
- The data is not actually loaded into the global R environment (see the Environment pane in RStudio).

```r
# inspect the object
str(flights)
```

```
## 'SparkDataFrame': 19 variables:
##  $ year          : chr "2013" "2013" "2013" "2013" "2013" "2013"
##  $ month         : chr "1" "1" "1" "1" "1" "1"
##  $ day           : chr "1" "1" "1" "1" "1" "1"
##  $ dep_time      : chr "517" "533" "542" "544" "554" "554"
##  $ sched_dep_time: chr "515" "529" "540" "545" "600" "558"
##  $ dep_delay     : chr "2" "4" "2" "-1" "-6" "-4"
##  $ arr_time      : chr "830" "850" "923" "1004" "812" "740"
##  $ sched_arr_time: chr "819" "830" "850" "1022" "837" "728"
##  $ arr_delay     : chr "11" "20" "33" "-18" "-25" "12"
##  $ carrier       : chr "UA" "UA" "AA" "B6" "DL" "UA"
##  $ flight        : chr "1545" "1714" "1141" "725" "461" "1696"
##  $ tailnum       : chr "N14228" "N24211" "N619AA" "N804JB" "N668DN" "N39463"
##  $ origin        : chr "EWR" "LGA" "JFK" "JFK" "LGA" "EWR"
##  $ dest          : chr "IAH" "IAH" "MIA" "BQN" "ATL" "ORD"
##  $ air_time      : chr "227" "227" "160" "183" "116" "150"
##  $ distance      : chr "1400" "1416" "1089" "1576" "762" "719"
##  $ hour          : chr "5" "5" "5" "5" "6" "5"
```

Data import and summary statistics

- By default, all variables have been imported as type `character`.
- Convert some columns to other data types with the `cast` function.

```
flights$dep_delay <- cast(flights$dep_delay, "double")
flights$dep_time <- cast(flights$dep_time, "double")
flights$arr_time <- cast(flights$arr_time, "double")
flights$arr_delay <- cast(flights$arr_delay, "double")
flights$air_time <- cast(flights$air_time, "double")
flights$distance <- cast(flights$distance, "double")
```

Data import and summary statistics

- Task: Compute average arrival delays per carrier for flights with a distance over 1000 miles.
- Variable selection and filtering of observations is implemented in `select()` and `filter()` (as in the `dplyr` package).

```
# filter
long_flights <- select(flights, "carrier", "year", "arr_delay", "distance")
long_flights <- filter(long_flights, long_flights$distance >= 1000)
head(long_flights)
```

```
##   carrier year arr_delay distance
## 1      UA 2013        11     1400
## 2      UA 2013        20     1416
## 3      AA 2013        33     1089
## 4      B6 2013       -18     1576
## 5      B6 2013        19     1065
## 6      B6 2013        -2     1028
```

Data import and summary statistics

- Summarize the arrival delays for the subset of long flights by carrier.
- This is the 'split-apply-combine' approach in SparkR! ¨

```
# aggregation: mean delay per carrier
long_flights_delays<- summarize(groupBy(long_flights, long_flights$carrier),
                      avg_delay = mean(long_flights$arr_delay))
head(long_flights_delays)
```

```
##   carrier  avg_delay
## 1      UA  3.2621897
## 2      AA  0.4957546
## 3      EV 15.6875637
## 4      B6  9.0364413
## 5      DL -0.2393537
## 6      OO -2.0000000
```

Data import and summary statistics

- Convert the result back into a usual `data.frame` (loaded in our current R session).
- Converting a `SparkDataFrame` back into a native R object means **all the data** stored in the RDDs constituting the `SparkDataFrame` object are **loaded into local RAM**!

```
# Convert result back into native R object
delays <- collect(long_flights_delays)
class(delays)
```

```
## [1] "data.frame"
```

```
delays
```

```
##    carrier  avg_delay
## 1       UA  3.2621897
## 2       AA  0.4957546
## 3       EV 15.6875637
## 4       B6  9.0364413
## 5       DL -0.2393537
## 6       OO -2.0000000
## 7       F9 21.9207048
## 8       US  0.5566964
```

Regression analysis

- Correlation study of what factors are associated with more or less arrival delay.
- Built-in ‘MLib’ library several high-level functions for regression analyses.

Regression analysis: comparison with native R

- First estimate a linear model with the usual R approach (all computed in the R environment).

```
# flights_r <- collect(flights) # very slow!
flights_r <- data.table::fread("../data/flights.csv", nrows = 300)


# specify the linear model
model1 <- arr_delay ~ dep_delay + distance
# fit the model with ols
fit1 <- lm(model1, flights_r)
# compute t-tests etc.
summary(fit1)
```

```
##
## Call:
## lm(formula = model1, data = flights_r)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -42.386  -9.965  -1.911   9.866  48.024
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.182662   1.676560  -0.109    0.913
## dep_delay    0.989553   0.017282  57.261   <2e-16 ***
```

Regression analysis: comparison with native R

- Compute essentially the same model estimate in `sparklyr`.
- Note that most regression models commonly used in traditional applied econometrics are also provided in `sparklyr` or `SparkR`.

```
library(sparklyr)

# connect with default configuration
sc <- spark_connect(master = "local",
                    version = "2.4.5")

# load data to spark
flights2 <- copy_to(sc, flights_r, "flights2")

# fit the model
fit1_spark <- ml_linear_regression(flights2, formula = model1)
# compute t-tests etc.
summary(fit1_spark)
```

```
## Deviance Residuals:
##     Min      1Q  Median      3Q     Max
## -42.386  -9.965  -1.911   9.866  48.024
##
## Coefficients:
##   (Intercept)     dep_delay      distance
## -0.1826622687  0.9895529018  0.0001139616
```

Regression analysis: comparison with native R

Alternative with `SparkR`:

```
# create SparkDataFrame
flights3 <- createDataFrame(flights_r)
# fit the model
fit2_spark <- spark.glm(formula = model1, data = flights3 , family="gaussian")
# compute t-tests etc.
summary(fit2_spark)
```

GPUs for Scientific Computing

GPUs for scientific computing

- Graphic Processing Units (GPUs).
- ‘Side product’ of the computer games industry.
- More demanding games needed better graphic cards (with faster GPUs).

GPUs

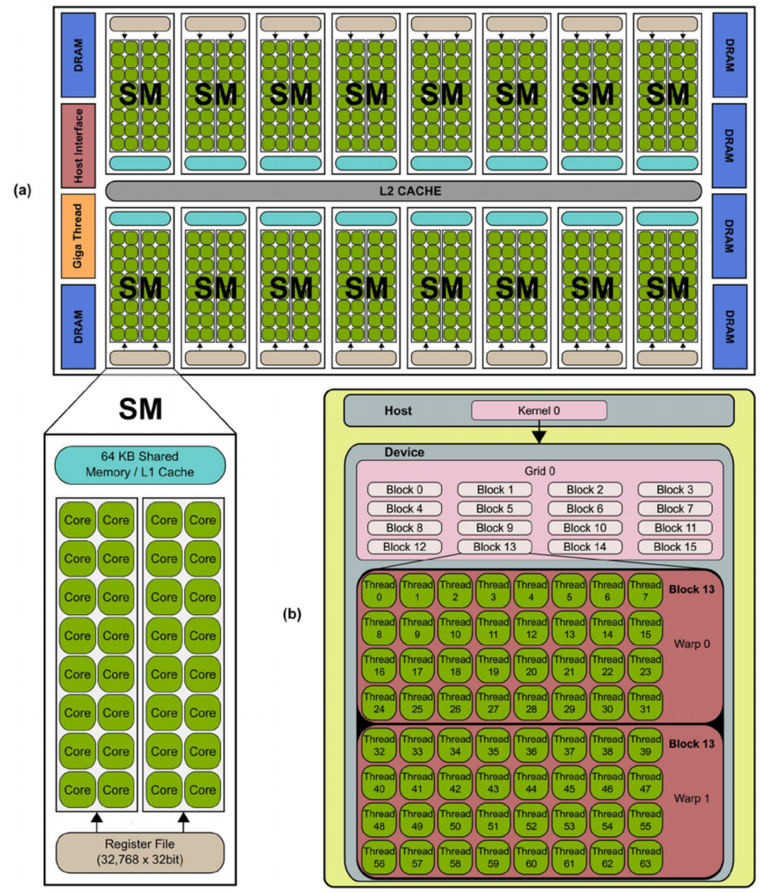
Why GPUs?

- Why not more powerful CPUs to deal with the more demanding PC games?
- CPUs: designed not only for **efficiency but also flexibility**.
- GPUs: designed to excel at computing graphics.
  - Highly parallel numerical floating point workloads.
  - Very useful in some core scientific computing tasks (see Fatahalian, Sugerman, and Hanrahan (2004))!

GPU characteristics

- Composed of several multiprocessor units.
- Each multiprocessor units has several cores.
- GPUs can perform computations with thousands of threads in parallel.

GPU characteristics



Typical NVIDIA GPU architecture (illustration and notes by Hernández et al. (2013)): The GPU is comprised of a set of Streaming MultiProcessors (SM). Each SM is comprised of several Stream Processor (SP) cores, as shown for the NVIDIA's Fermi architecture (a). The GPU resources are controlled by the programmer through the CUDA programming model, shown in (b).

Challenges to using GPUs for scientific computing

- Different hardware architecture, different low-level programming model.
- Good understanding of hardware needed.
- But, more and more high-level APIs available (e.g., in tensorflow/keras).

GPUs in R

Example I: Matrix multiplication comparison (gpuR)

- gpuR: basic R functions to compute with GPUs from within the R environment.
- Example: compare the performance of the CPU with the GPU based on a matrix multiplication exercise.
  - (For a large $N \times P$ matrix X, we want to compute $X^t X$.)

Example I: Matrix multiplication comparison (gpuR)

```
# load package
library(bench)
library(gpuR)
```

```
## Number of platforms: 1
## - platform: NVIDIA Corporation: OpenCL 1.2 CUDA 11.2.136
##   - context device index: 0
##     - GeForce GTX 1650
## checked all devices
## completed initialization
```

Example I: Matrix multiplication comparison (**gpuR**)

Initiate a large matrix filled with pseudo random numbers (N observations and P variables).

```
# initiate dataset with pseudo random numbers
N <- 10000  # number of observations
P <- 100 # number of variables
X <- matrix(rnorm(N * P, 0, 1), nrow = N, ncol =P)
```

Example I: Matrix multiplication comparison (gpuR)

Prepare for GPU computation.

```
# prepare GPU-specific objects/settings
gpuX <- gpuMatrix(X, type = "float")  # point GPU to matrix (matrix stored in non-GPU memory)
vclX <- vclMatrix(X, type = "float")  # transfer matrix to GPU (matrix stored in GPU memory)
```

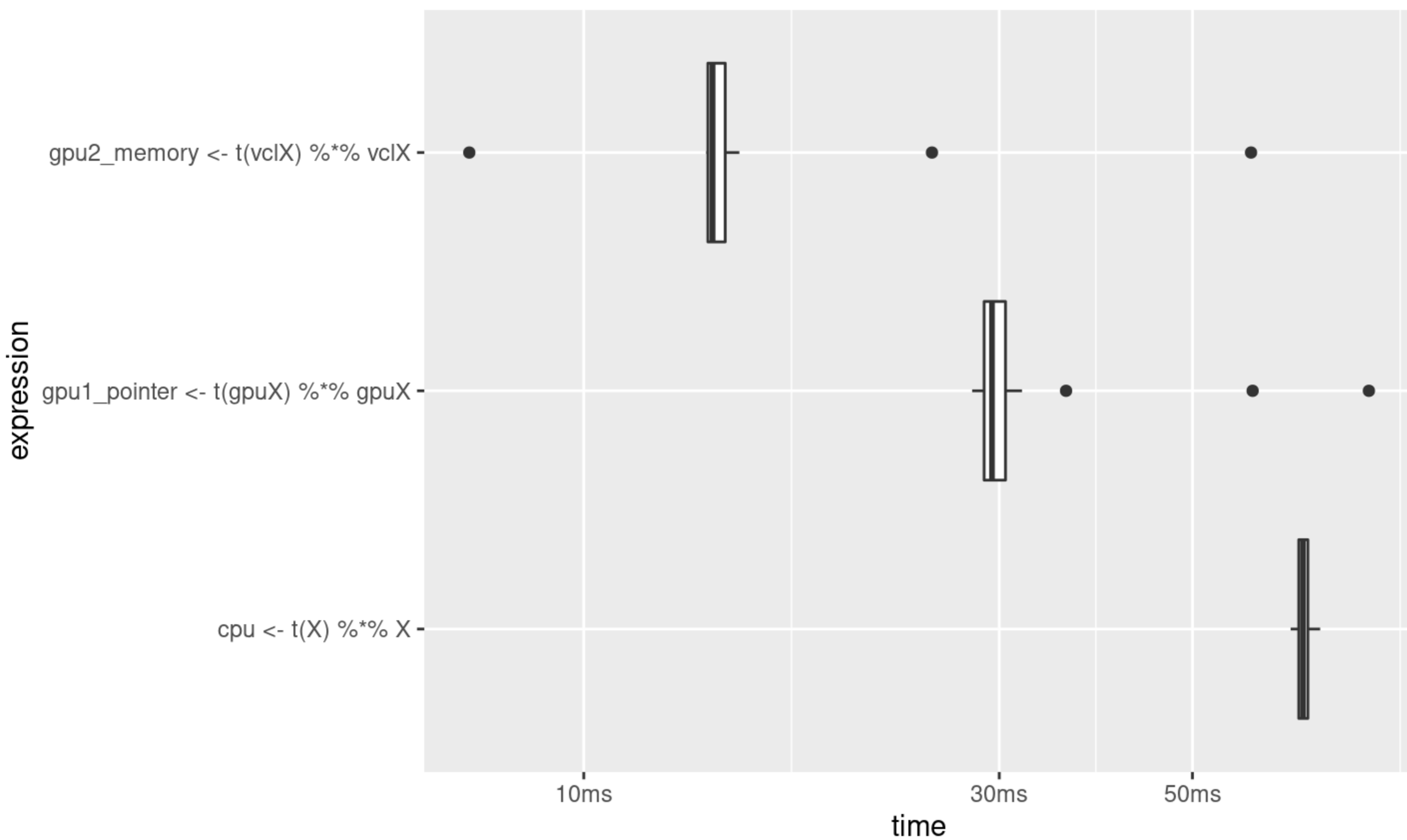
Example I: Matrix multiplication comparison (gpuR)

Now we run the three examples: 1) using the CPU, 2) computing on the GPU but using CPU memory, 3) computing on the GPU and using GPU memory.

```
# compare three approaches
gpu_cpu <- bench::mark(

  # compute with CPU
  cpu <- t(X) %*% X,

  # GPU version, GPU pointer to CPU memory (gpuMatrix is simply a pointer)
  gpu1_pointer <- t(gpuX) %*% gpuX,

  # GPU version, in GPU memory (vclMatrix formation is a memory transfer)
  gpu2_memory <- t(vclX) %*% vclX,

check = FALSE, memory=FALSE, min_iterations = 20)
```

Example I: Matrix multiplication comparison (gpuR)

```
plot(gpu_cpu, type = "boxplot")
```



Example II: GPUs and Machine Learning

- Training deep learning (DL) models depends on tensor (matrix) multiplications.
- DL typically relies on highly parallelized computing based on GPUs.
- Software to build and train deep neural nets (tensorflow, and the high-level Keras API) make it comparatively simple to use GPUs.

Tensorflow/Keras example: predict housing prices

In this example we train a simple sequential model with two hidden layers in order to predict the median value of owner-occupied homes (in USD 1,000) in the Boston area (data are from the 1970s). The original data and a detailed description can be found here. The example follows closely this keras tutorial published by RStudio.

Tensorflow/Keras example: predict housing prices

```r
# load packages
library(keras)
library(tibble)
library(ggplot2)
library(tfdatasets)
```

```
##
## Attaching package: 'tfdatasets'

## The following object is masked from 'package:SparkR':
##
##     contains
```

```r
# load data
boston_housing <- dataset_boston_housing()
str(boston_housing)
```

```
## List of 2
##  $ train:List of 2
##   ..$ x: num [1:404, 1:13] 1.2325 0.0218 4.8982 0.0396 3.6931 ...
##   ..$ y: num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 14.4 ...
##  $ test :List of 2
##   ..$ x: num [1:102, 1:13] 18.0846 0.1233 0.055 1.2735 0.0715 ...
##   ..$ y: num [1:102(1d)] 7.2 18.8 19 27 22.2 24.5 31.2 22.9 20.5 23.2 ...
```

Training and test dataset

- First split the data into a training set and a test set.
- Reason: Monitor the out-of-sample performance of the trained model!
  - (Deep) neural nets are often susceptible to over-fitting.
  - Validity checks based on the test sample are often an integral part of modelling with tensorflow/keras.

```
# assign training and test data/labels
c(train_data, train_labels) %<-% boston_housing$train
c(test_data, test_labels) %<-% boston_housing$test
```

Prepare training data

In order to better understand and interpret the dataset we add the original variable names, and convert the dataset to a `tibble`.

```
library(dplyr)

column_names <- c('CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
                  'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT')

train_df <- train_data %>%
  as_tibble(.name_repair = "minimal") %>%
  setNames(column_names) %>%
  mutate(label = train_labels)

test_df <- test_data %>%
  as_tibble(.name_repair = "minimal") %>%
  setNames(column_names) %>%
  mutate(label = test_labels)
```

Inspect training data

Next, we have a close look at the data. Note the usage of the term ‘label’ for what is usually called the ‘dependent variable’ in econometrics.

```
# check example data dimensions and content
paste0("Training entries: ", length(train_data), ", labels: ", length(train_labels))
```

```
## [1] "Training entries: 5252, labels: 404"
```

```
summary(train_data)
```

```
##        V1                V2               V3              V4                V5
##  Min.   : 0.00632   Min.   :  0.00   Min.   : 0.46   Min.   :0.00000   Min.   :0.3850
##  1st Qu.: 0.08144   1st Qu.:  0.00   1st Qu.: 5.13   1st Qu.:0.00000   1st Qu.:0.4530
##  Median : 0.26888   Median :  0.00   Median : 9.69   Median :0.00000   Median :0.5380
##  Mean   : 3.74511   Mean   : 11.48   Mean   :11.10   Mean   :0.06188   Mean   :0.5574
##  3rd Qu.: 3.67481   3rd Qu.: 12.50   3rd Qu.:18.10   3rd Qu.:0.00000   3rd Qu.:0.6310
##  Max.   :88.97620   Max.   :100.00   Max.   :27.74   Max.   :1.00000   Max.   :0.8710
##        V6               V7               V8               V9               V10              V11
##  Min.   :3.561   Min.   :  2.90   Min.   : 1.130   Min.   : 1.000   Min.   :188.0   Min.   :12.60
##  1st Qu.:5.875   1st Qu.: 45.48   1st Qu.: 2.077   1st Qu.: 4.000   1st Qu.:279.0   1st Qu.:17.23
##  Median :6.199   Median : 78.50   Median : 3.142   Median : 5.000   Median :330.0   Median :19.10
##  Mean   :6.267   Mean   : 69.01   Mean   : 3.740   Mean   : 9.441   Mean   :405.9   Mean   :18.48
##  3rd Qu.:6.609   3rd Qu.: 94.10   3rd Qu.: 5.118   3rd Qu.:24.000   3rd Qu.:666.0   3rd Qu.:20.20
##  Max.   :8.725   Max.   :100.00   Max.   :10.710   Max.   :24.000   Max.   :711.0   Max.   :22.00
##       V12              V13
```

Normalize features

- The dataset contains variables ranging from per capita crime rate to indicators for highway access (different units, different scales).
- Not per se a problem, but fitting is more efficient when all features are normalized.

Normalize features

```
spec <- feature_spec(train_df, label ~ . ) %>%
  step_numeric_column(all_numeric(), normalizer_fn = scaler_standard()) %>%
  fit()

layer <- layer_dense_features(
  feature_columns = dense_features(spec),
  dtype = tf$float32
)
layer(train_df)
```

```
## tf.Tensor(
## [[ 0.81205493  0.44752213 -0.2565147  ... -0.1762239  -0.59443307
##   -0.48301655]
##  [-1.9079947   0.43137115 -0.2565147  ...  1.8920003  -0.34800112
##    2.9880793 ]
##  [ 1.1091131   0.2203439  -0.2565147  ... -1.8274226   1.563349
##   -0.48301655]
##  ...
##  [-1.6359899   0.07934052 -0.2565147  ... -0.3326088  -0.61246467
##    0.9895695 ]
##  [ 1.0554279  -0.98642045 -0.2565147  ... -0.7862657  -0.01742171
##   -0.48301655]
##  [-1.7970455   0.23288251 -0.2565147  ...  0.47467488 -0.84687555
##    2.0414166 ]], shape=(404, 13), dtype=float32)
```

Model specification

We specify the model as a linear stack of layers:

- The input (all 13 explanatory variables).
- Two densely connected hidden layers (each with a 64-dimensional output space).
- The one-dimensional output layer (the 'dependent variable').

```r
# Create the model
# model specification
input <- layer_input_from_dataset(train_df %>% select(-label))

output <- input %>%
  layer_dense_features(dense_features(spec)) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1)

model <- keras_model(input, output)
```

Training configuration

In order to fit the model, we first have to 'compile' it (configure it for training):

- Set the parameters that will guide the training/optimization procedure.
  - Mean squared errors loss function (`mse`) typically used for regressions.
  - RMSProp optimizer to find the minimum loss.

```
# compile the model
model %>%
  compile(
    loss = "mse",
    optimizer = optimizer_rmsprop(),
    metrics = list("mean_absolute_error")
  )
```

Training configuration

Now we can get a summary of the model we are about to fit to the data.

```
# get a summary of the model
model
```

```
## Model
## Model: "model"
## ________________________________________________________________________________________________________
## Layer (type)                     Output Shape           Param #     Connected to
## ========================================================================================================
## AGE (InputLayer)                 [(None,)]              0
## ________________________________________________________________________________________________________
## B (InputLayer)                   [(None,)]              0
## ________________________________________________________________________________________________________
## CHAS (InputLayer)                [(None,)]              0
## ________________________________________________________________________________________________________
## CRIM (InputLayer)                [(None,)]              0
## ________________________________________________________________________________________________________
## DIS (InputLayer)                 [(None,)]              0
## ________________________________________________________________________________________________________
## INDUS (InputLayer)               [(None,)]              0
## ________________________________________________________________________________________________________
## LSTAT (InputLayer)               [(None,)]              0
## ________________________________________________________________________________________________________
## NOX (InputLayer)                 [(None,)]              0
## ________________________________________________________________________________________________________
```

Monitoring training progress

- Set number of epochs.

```
# Set max. number of epochs
epochs <- 500
```

Fit (train) the model

```
# Fit the model and store training stats

history <- model %>% fit(
  x = train_df %>% select(-label),
  y = train_df$label,
  epochs = epochs,
  validation_split = 0.2,
  verbose = 0
)


plot(history)
```

Parallelization: A word of caution

Why not always use GPUs for parallel tasks in scientific computing?

- Whether a GPU implementation is faster, depends on many factors.
- Also, proper implementation of parallel tasks (either on GPUs or CPUs) can be very tricky (and a lot of work).

References

Fatahalian, K., J. Sugerman, and P. Hanrahan. 2004. "Understanding the Efficiency of Gpu Algorithms for Matrix-Matrix Multiplication." In **Proceedings of the Acm Siggraph/Eurographics Conference on Graphics Hardware**, 133–37. HWWS '04. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1058129.1058148.

Hernández, Moisés, Ginés D. Guerrero, José M. Cecilia, José M. García, Alberto Inuggi, Saad Jbabdi, Timothy E. J. Behrens, and Stamatios N. Sotiropoulos. 2013. "Accelerating Fibre Orientation Estimation from Diffusion Weighted Magnetic Resonance Imaging Using Gpus." **PLOS ONE** 8 (4): 1–13. https://doi.org/10.1371/journal.pone.0061892.