A Minimal rTorch Tutorial

Alfonso R. Reyes

2019-09-20

Contents

Prerequisites

You need two things to get `rTorch` working:

1. Install Python Anaconda. Preferrably, for 64-bits, and above Python 3.6+.
2. Install R, Rtools and RStudio.
3. Install `rTorch` from CRAN or GitHub.

   Note. It is not mandatory to have a previously created `Python` environment with `Anaconda`, where `PyTorch` and `TorchVision` have already been installed. This step is optional. You could also get it installed directly from the `R` console, in very similar fashion as in R-TensorFlow using the function `install_pytorch`.

This book is available online via GitHub Pages, or you can also build it from source from its repository.

Installation

`rTorch` is available via CRAN or GitHub.

The **rTorch** package can be installed from CRAN or Github.

From CRAN:

```
install.packages("rTorch")
```

From GitHub, install `rTorch` with:

```
devtools::install_github("f0nzie/rTorch")
```

Python Anaconda

Before start running `rTorch`, install a Python Anaconda environment first.

Example

1. Create a `conda` environment from the terminal with `conda create -n myenv python=3.7`
2. Activate the new environment with `conda activate myenv`
3. Install the `PyTorch` related packages with:

```
conda install python=3.6.6 pytorch-cpu torchvision-cpu matplotlib
pandas -c pytorch
```

The last part `-c pytorch` specifies the conda channel to download the PyTorch packages. Your installation may not work if you don't indicate the channel.

Now, you can load `rTorch` in R or RStudio.

Automatic installation

I use the idea from automatic installation in `r-tensorflow`, to create the function `rTorch::install_pytorch()`. This function will allow you to install a `conda` environment complete with all `PyTorch` requirements.

> **Note.** `matplotlib` and `pandas` are not really necessary for `rTorch` to work, but I was asked if `matplotlib` or `pandas` would work with `PyTorch`. So, I decided to install them for testing and experimentation. They both work.

Part I

Getting Started

Chapter 1

Introduction

1.1 Motivation

Why do we want a package of something that is already working well, such as PyTorch?

There are several reasons, but the main one is to bring another machine learning framework to R. Probably it just me but I feel PyTorch very comfortable to work with. Feels pretty much like everything else in Python. I have tried other frameworks in R. The closest that matches a natural language like PyTorch, is MXnet. Unfortunately, it is the hardest to install and maintain after updates.

Yes. I could have worked directly with PyTorch in a native Python environment, such as Jupyter or PyCharm but it very hard to quit **RMarkdown** once you get used to it. It is the real thing in regards to literate programming. It does not only contributes to improving the quality of the code but establishes a workflow for a better understanding of the subject by your intended readers (Knuth, 1983), in what is been called the *literate programming paradigm* (Cordes and Brown, 1991).

This has the additional benefit of giving the ability to write combination of Python and R code together in the same document. There will times when it is better to create a class in Python; and other times where R will be more convenient to handle a data structure.

1.2 How do we start using `rTorch`

Start using `rTorch` is very simple. After installing the minimum system requirements, you just call it with:

```
library(rTorch)

transforms  <- torchvision$transforms
```

There are several way of testing that `rTorch` is up and running. Let's see some of them:

1.2.1 Getting the PyTorch version

```
rTorch::torch_version()
#> [1] "1.1"
```

1.2.2 PyTorch configuration

This will show the PyTorch version and the current version of Python installed, as well as the paths where they reside.

```
rTorch::torch_config()
#> PyTorch v1.1.0 (~/anaconda3/envs/r-torch/lib/python3.6/site-packages/torch)
#> Python v3.6 (~/anaconda3/envs/r-torch/bin/python)
```

1.3 What can you do with `rTorch`

Practically, you can do everything you could with PyTorch within the R ecosystem. Additionally to the `rTorch` module, from where you can extract methods, functions and classes, there are available two more modules: `torchvision` and `np`, which is short for `numpy`.

1.3.1 The `torchvision` module

This is an example of using the `torchvision` module. With `torchvision` we could download any of the datasets available. In this case, we will be downloading the training dataset of the **MNIST** handwritten digits. There are 60,000 images in the training set.

```
local_folder <- '../datasets/mnist_digits'
train_dataset = torchvision$datasets$MNIST(root = local_folder,
                                           train = TRUE,
                                           transform = transforms$ToTensor(),
                                           download = TRUE)
```

```
train_dataset
#> Dataset MNIST
#>     Number of datapoints: 60000
#>     Root location: ../datasets/mnist_digits
#>     Split: Train
```

You can do similarly for the `test` dataset if you set the flag `train = FALSE`. The `test` dataset has only 10,000 images.

```
test_dataset = torchvision$datasets$MNIST(root = local_folder,
                                          train = FALSE,
                                          transform = transforms$ToTensor())
test_dataset
#> Dataset MNIST
#>     Number of datapoints: 10000
#>     Root location: ../datasets/mnist_digits
#>     Split: Test
```

1.3.2 np: the numpy module

`numpy` is automaticaly installed when `PyTorch` is. There is some interdependence between both. Anytime that we need to do some transformation that is not available in `PyTorch`, we will use `numpy`.

There are several operations that we could perform with `numpy`:

Create an array

```
# do some array manipulations with NumPy
a <- np$array(c(1:4))
a
#> [1] 1 2 3 4
```

```
np$reshape(np$arange(0, 9), c(3L, 3L))
#>      [,1] [,2] [,3]
#> [1,]    0    1    2
#> [2,]    3    4    5
#> [3,]    6    7    8
```

```
np$array(list(
             list(73, 67, 43),
             list(87, 134, 58),
             list(102, 43, 37),
             list(73, 67, 43),
             list(91, 88, 64),
```

```
            list(102, 43, 37),
            list(69, 96, 70),
            list(91, 88, 64),
            list(102, 43, 37),
            list(69, 96, 70)
          ), dtype='float32')
#>       [,1] [,2] [,3]
#>  [1,]   73   67   43
#>  [2,]   87  134   58
#>  [3,]  102   43   37
#>  [4,]   73   67   43
#>  [5,]   91   88   64
#>  [6,]  102   43   37
#>  [7,]   69   96   70
#>  [8,]   91   88   64
#>  [9,]  102   43   37
#> [10,]   69   96   70
```

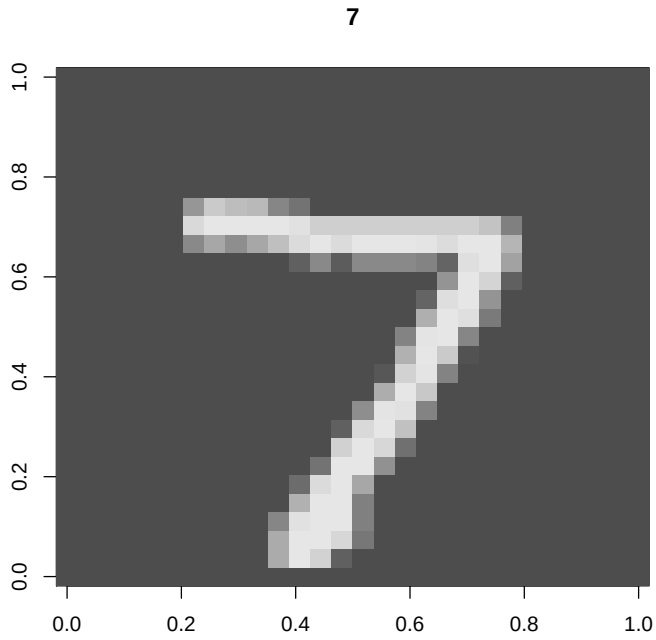
Reshape an array

For the same `test` dataset that we loaded above, we will show the image of the handwritten digit and its label or class. Before plotting the image, we need to:

1. Extract the image and label from the dataset
2. Convert the tensor to a numpy array
3. Reshape the tensor as a 2D array
4. Plot the digit and its label

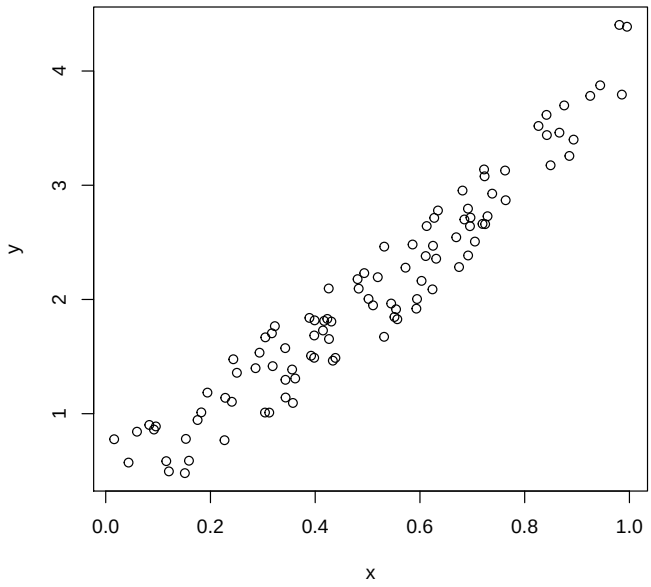
```
rotate <- function(x) t(apply(x, 2, rev))   # function to rotate the matrix

# label for the image
label <- test_dataset[0][[2]]
label
#> [1] 7

# convert tensor to numpy array
.show_img <- test_dataset[0][[1]]$numpy()
dim(.show_img)
#> [1]  1 28 28

# reshape 3D array to 2D
show_img <- np$reshape(.show_img, c(28L, 28L))
dim(show_img)
#> [1] 28 28
```

```
# show in grays and rotate
image(rotate(show_img), col = gray.colors(64))
title(label)
```



Generate a random array

```
# set the seed
np$random$seed(123L)
# generate a random array
x = np$random$rand(100L)
# calculate the y array
y = np$sin(x) * np$power(x, 3L) + 3L * x + np$random$rand(100L) * 0.8

plot(x, y)
```



Convert a `numpy` array to a PyTorch tensor

This is a very common operation that I have seen in examples using PyTorch. Creating the array in `numpy`. and then convert it to a tensor.

```
# input array
x = np$array(rbind(
              c(0,0,1),
              c(0,1,1),
              c(1,0,1),
              c(1,1,1)))

# the numpy array
x
#>      [,1] [,2] [,3]
#> [1,]    0    0    1
#> [2,]    0    1    1
#> [3,]    1    0    1
#> [4,]    1    1    1
```

```
# convert the numpy array to float
X <- np$float32(x)
# convert the numpy array to a float tensor
X <- torch$FloatTensor(X)
X
#> tensor([[0., 0., 1.],
```

```
#>         [0., 1., 1.],
#>         [1., 0., 1.],
#>         [1., 1., 1.]])
```

1.3.3 Python built-in functions

To access the Python built-in functions we make use of the package `reticulate` and the function `import_builtins()`.

```
py_bi <- import_builtins()
```

Length of a dataset

```
py_bi$len(train_dataset)
#> [1] 60000
py_bi$len(test_dataset)
#> [1] 10000
```

Iterators

```
# iterate through training dataset
enum_train_dataset <- py_bi$enumerate(train_dataset)

cat(sprintf("%8s %8s \n", "index", "label"))
#>    index    label
for (i in 1:py_bi$len(train_dataset)) {
    obj <- reticulate::iter_next(enum_train_dataset)
    idx   <- obj[[1]]        # index number
    cat(sprintf("%8d %5d \n", idx, obj[[2]][[2]]))

    if (i >= 100) break   # print only 100 labels
}
#>        0     5
#>        1     0
#>        2     4
#>        3     1
#>        4     9
#>        5     2
#>        6     1
#>        7     3
#>        8     1
#>        9     4
```

```
#>         10      3
#>         11      5
#>         12      3
#>         13      6
#>         14      1
#>         15      7
#>         16      2
#>         17      8
#>         18      6
#>         19      9
#>         20      4
#>         21      0
#>         22      9
#>         23      1
#>         24      1
#>         25      2
#>         26      4
#>         27      3
#>         28      2
#>         29      7
#>         30      3
#>         31      8
#>         32      6
#>         33      9
#>         34      0
#>         35      5
#>         36      6
#>         37      0
#>         38      7
#>         39      6
#>         40      1
#>         41      8
#>         42      7
#>         43      9
#>         44      3
#>         45      9
#>         46      8
#>         47      5
#>         48      9
#>         49      3
#>         50      3
#>         51      0
#>         52      7
#>         53      4
#>         54      9
```

```
#>         55      8
#>         56      0
#>         57      9
#>         58      4
#>         59      1
#>         60      4
#>         61      4
#>         62      6
#>         63      0
#>         64      4
#>         65      5
#>         66      6
#>         67      1
#>         68      0
#>         69      0
#>         70      1
#>         71      7
#>         72      1
#>         73      6
#>         74      3
#>         75      0
#>         76      2
#>         77      1
#>         78      1
#>         79      7
#>         80      9
#>         81      0
#>         82      2
#>         83      6
#>         84      7
#>         85      8
#>         86      3
#>         87      9
#>         88      0
#>         89      4
#>         90      6
#>         91      7
#>         92      4
#>         93      6
#>         94      8
#>         95      0
#>         96      7
#>         97      8
#>         98      3
#>         99      1
```

Types and instances

```
# get the class of the object
py_bi$type(train_dataset)
#> <class 'torchvision.datasets.mnist.MNIST'>

# is train_dataset a torchvision dataset class
py_bi$isinstance(train_dataset, torchvision$datasets$mnist$MNIST)
#> [1] TRUE
```

Chapter 2

rTorch vs PyTorch: What's different

This chapter will explain the main differences between `PyTorch` and `rTorch`. Most of the things work directly in `PyTorch` but we need to be aware of some minor differences when working with rTorch.

Here is a review of existing methods.

```
library(rTorch)
```

2.1 Calling objects from PyTorch

We use the dollar sign or `$` to call a class, function or method from the rTorch modules. In this case, from `torch` module:

```
torch$tensor
#> <built-in method tensor of type>
```

2.2 Call a module from PyTorch

```
# these are the equivalents of import module
nn          <- torch$nn
transforms  <- torchvision$transforms
dsets       <- torchvision$datasets
```

Then we can proceed to extract classes, methods and functions from the `nn`, `transforms`, and `dsets` objects.

2.3 Show the attributes (methods) of a class or PyTorch object

Sometimes we are interested in knowing the internal components of a class. In that case, we use the reticulate function `py_list_attributes()`.

```
local_folder <- '../datasets/mnist_digits'
train_dataset = torchvision$datasets$MNIST(root = local_folder,
                                           train = TRUE,
                                           transform = transforms$ToTensor(),
                                           download = TRUE)

train_dataset
#> Dataset MNIST
#>     Number of datapoints: 60000
#>     Root location: ../datasets/mnist_digits
#>     Split: Train
```

```
reticulate::py_list_attributes(train_dataset)
#>  [1] "__add__"                "__class__"
#>  [3] "__delattr__"            "__dict__"
#>  [5] "__dir__"                "__doc__"
#>  [7] "__eq__"                 "__format__"
#>  [9] "__ge__"                 "__getattribute__"
#> [11] "__getitem__"            "__gt__"
#> [13] "__hash__"               "__init__"
#> [15] "__init_subclass__"      "__le__"
#> [17] "__len__"                "__lt__"
#> [19] "__module__"             "__ne__"
#> [21] "__new__"                "__reduce__"
#> [23] "__reduce_ex__"          "__repr__"
#> [25] "__setattr__"            "__sizeof__"
#> [27] "__str__"                "__subclasshook__"
#> [29] "__weakref__"            "_check_exists"
#> [31] "_format_transform_repr" "_repr_indent"
#> [33] "class_to_idx"           "classes"
#> [35] "data"                   "download"
#> [37] "extra_repr"             "extract_gzip"
#> [39] "processed_folder"       "raw_folder"
#> [41] "root"                   "target_transform"
#> [43] "targets"                "test_data"
#> [45] "test_file"              "test_labels"
#> [47] "train"                  "train_data"
#> [49] "train_labels"           "training_file"
#> [51] "transform"              "transforms"
```

```
#> [53] "urls"
```

Knowing the internal methods of a class could be useful when we want to refer to a specific property of such class. For example, from the list above, we know that the object `train_dataset` has an attribute `__len__`. We can call it like this:

```
train_dataset$`__len__`()
#> [1] 60000
```

2.4 Enumeration

```
x_train = array(c(3.3, 4.4, 5.5, 6.71, 6.93, 4.168,
                  9.779, 6.182, 7.59, 2.167, 7.042,
                  10.791, 5.313, 7.997, 3.1), dim = c(15,1))

x_train <- r_to_py(x_train)
x_train <- torch$from_numpy(x_train)          # convert to tensor
x_train <- x_train$type(torch$FloatTensor)    # make it a a FloatTensor

x_train
#> tensor([[ 3.3000],
#>         [ 4.4000],
#>         [ 5.5000],
#>         [ 6.7100],
#>         [ 6.9300],
#>         [ 4.1680],
#>         [ 9.7790],
#>         [ 6.1820],
#>         [ 7.5900],
#>         [ 2.1670],
#>         [ 7.0420],
#>         [10.7910],
#>         [ 5.3130],
#>         [ 7.9970],
#>         [ 3.1000]])
```

```
x_train$nelement()    # number of elements in the tensor
#> [1] 15
```

2.5 How to iterate

2.5.1 Using `enumerate` and `iterate`

```
py = import_builtins()

enum_x_train = py$enumerate(x_train)
enum_x_train
#> <enumerate>

py$len(x_train)
#> [1] 15
```

```
xit = iterate(enum_x_train, simplify = TRUE)
xit
#> [[1]]
#> [[1]][[1]]
#> [1] 0
#>
#> [[1]][[2]]
#> tensor([3.3000])
#>
#>
#> [[2]]
#> [[2]][[1]]
#> [1] 1
#>
#> [[2]][[2]]
#> tensor([4.4000])
#>
#>
#> [[3]]
#> [[3]][[1]]
#> [1] 2
#>
#> [[3]][[2]]
#> tensor([5.5000])
#>
#>
#> [[4]]
#> [[4]][[1]]
#> [1] 3
#>
#> [[4]][[2]]
#> tensor([6.7100])
```

```
#>
#>
#> [[5]]
#> [[5]][[1]]
#> [1] 4
#>
#> [[5]][[2]]
#> tensor([6.9300])
#>
#>
#> [[6]]
#> [[6]][[1]]
#> [1] 5
#>
#> [[6]][[2]]
#> tensor([4.1680])
#>
#>
#> [[7]]
#> [[7]][[1]]
#> [1] 6
#>
#> [[7]][[2]]
#> tensor([9.7790])
#>
#>
#> [[8]]
#> [[8]][[1]]
#> [1] 7
#>
#> [[8]][[2]]
#> tensor([6.1820])
#>
#>
#> [[9]]
#> [[9]][[1]]
#> [1] 8
#>
#> [[9]][[2]]
#> tensor([7.5900])
#>
#>
#> [[10]]
#> [[10]][[1]]
#> [1] 9
```

```
#>
#> [[10]][[2]]
#> tensor([2.1670])
#>
#>
#> [[11]]
#> [[11]][[1]]
#> [1] 10
#>
#> [[11]][[2]]
#> tensor([7.0420])
#>
#>
#> [[12]]
#> [[12]][[1]]
#> [1] 11
#>
#> [[12]][[2]]
#> tensor([10.7910])
#>
#>
#> [[13]]
#> [[13]][[1]]
#> [1] 12
#>
#> [[13]][[2]]
#> tensor([5.3130])
#>
#>
#> [[14]]
#> [[14]][[1]]
#> [1] 13
#>
#> [[14]][[2]]
#> tensor([7.9970])
#>
#>
#> [[15]]
#> [[15]][[1]]
#> [1] 14
#>
#> [[15]][[2]]
#> tensor([3.1000])
```

2.5.2 Using a `for-loop` to iterate

```
# reset the iterator
enum_x_train = py$enumerate(x_train)

for (i in 1:py$len(x_train)) {
    obj <- iter_next(enum_x_train)    # next item
    cat(obj[[1]], "\t")     # 1st part or index
    print(obj[[2]])         # 2nd part or tensor
}
#> 0    tensor([3.3000])
#> 1    tensor([4.4000])
#> 2    tensor([5.5000])
#> 3    tensor([6.7100])
#> 4    tensor([6.9300])
#> 5    tensor([4.1680])
#> 6    tensor([9.7790])
#> 7    tensor([6.1820])
#> 8    tensor([7.5900])
#> 9    tensor([2.1670])
#> 10   tensor([7.0420])
#> 11   tensor([10.7910])
#> 12   tensor([5.3130])
#> 13   tensor([7.9970])
#> 14   tensor([3.1000])
```

> We will find very frequently this kind of iterators when we read a dataset using `torchvision`. There are different ways to iterate through these objects.

2.6 Zero gradient

The zero gradient was one of the most difficult to implement in R if we don't pay attention to the content of the objects carrying the weights and biases. This happens when the algorithm written in PyTorch is not immediately translatable to rTorch. This can be appreciated in this example.

2.6.1 Version in Python

```
import numpy as np
import torch
```

```
torch.manual_seed(0)  # reproducible

# Input (temp, rainfall, humidity)
#> <torch._C.Generator object at 0x7f631a5cae10>
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')

# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119]], dtype='float32')


# Convert inputs and targets to tensors
inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)

# random weights and biases
w = torch.randn(2, 3, requires_grad=True)
b = torch.randn(2, requires_grad=True)

# function for the model
def model(x):
  wt = w.t()
  mm = x @ w.t()
  return x @ w.t() + b       # @ represents matrix multiplication in PyTorch

# MSE loss function
def mse(t1, t2):
  diff = t1 - t2
  return torch.sum(diff * diff) / diff.numel()

# Running all together
# Train for 100 epochs
for i in range(100):
  preds = model(inputs)
  loss = mse(preds, targets)
  loss.backward()
  with torch.no_grad():
    w -= w.grad * 0.00001
```

```
    b -= b.grad * 0.00001
    w_gz = w.grad.zero_()
    b_gz = b.grad.zero_()

# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
print("Loss: ", loss)

# predictions
#> Loss:  tensor(1270.1234, grad_fn=<DivBackward0>)
print("\nPredictions:")
#>
#> Predictions:
preds

# Targets
#> tensor([[ 69.3122,  80.2639],
#>         [ 73.7528,  97.2381],
#>         [118.3933, 124.7628],
#>         [ 89.6111,  93.0286],
#>         [ 47.3014,  80.6467]], grad_fn=<AddBackward0>)
print("\nTargets:")
#>
#> Targets:
targets
#> tensor([[ 56.,  70.],
#>         [ 81., 101.],
#>         [119., 133.],
#>         [ 22.,  37.],
#>         [103., 119.]])
```

2.6.2 Version in R

```
library(rTorch)

torch$manual_seed(0)
#> <torch._C.Generator>

device = torch$device('cpu')
# Input (temp, rainfall, humidity)
inputs = np$array(list(list(73, 67, 43),
                       list(91, 88, 64),
```

```
                  list(87, 134, 58),
                  list(102, 43, 37),
                  list(69, 96, 70)), dtype='float32')

# Targets (apples, oranges)
targets = np$array(list(list(56, 70),
                    list(81, 101),
                    list(119, 133),
                    list(22, 37),
                    list(103, 119)), dtype='float32')


# Convert inputs and targets to tensors
inputs = torch$from_numpy(inputs)
targets = torch$from_numpy(targets)

# random numbers for weights and biases. Then convert to double()
torch$set_default_dtype(torch$float64)

w = torch$randn(2L, 3L, requires_grad=TRUE) #$double()
b = torch$randn(2L, requires_grad=TRUE) #$double()

model <- function(x) {
  wt <- w$t()
  return(torch$add(torch$mm(x, wt), b))
}

# MSE loss
mse = function(t1, t2) {
  diff <- torch$sub(t1, t2)
  mul <- torch$sum(torch$mul(diff, diff))
  return(torch$div(mul, diff$numel()))
}

# Running all together
# Adjust weights and reset gradients
for (i in 1:100) {
  preds = model(inputs)
  loss = mse(preds, targets)
  loss$backward()
  with(torch$no_grad(), {
    w$data <- torch$sub(w$data, torch$mul(w$grad, torch$scalar_tensor(1e-5)))
    b$data <- torch$sub(b$data, torch$mul(b$grad, torch$scalar_tensor(1e-5)))

    w$grad$zero_()
```

```
    b$grad$zero_()
  })
}

# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
cat("Loss: "); print(loss)
#> Loss:
#> tensor(1270.1237, grad_fn=<DivBackward0>)

# predictions
cat("\nPredictions:\n")
#>
#> Predictions:
preds
#> tensor([[ 69.3122,  80.2639],
#>         [ 73.7528,  97.2381],
#>         [118.3933, 124.7628],
#>         [ 89.6111,  93.0286],
#>         [ 47.3013,  80.6467]], grad_fn=<AddBackward0>)

# Targets
cat("\nTargets:\n")
#>
#> Targets:
targets
#> tensor([[ 56.,  70.],
#>         [ 81., 101.],
#>         [119., 133.],
#>         [ 22.,  37.],
#>         [103., 119.]])
```

Notice that while in Python, the tensor operation, gradient of the weights times the **Learning Rate**, is:

$$w = -w + \nabla w \; \alpha$$

is a very straight forwward and clean code:

```
w -= w.grad * 1e-5
```

In R shows a little bit more convoluted:

```
w$data <- torch$sub(w$data, torch$mul(w$grad, torch$scalar_tensor(1e-5)))
```

2.7 Transform a tensor

Explain how transform a tensor back and forth to `numpy`. Why is this important?

2.8 Build a model class

PyTorch classes cannot not directly instantiated from `R`. We need an intermediate step to create a class. For this, we use `reticulate` functions that will read the class implementation in `Python` code.

2.8.1 Example 1

2.8.2 Example 2

2.9 Convert a tensor to `numpy` object

This is a frequent operation. I have found that this is necessary when

- a numpy function is not implemented in PyTorch
- We need to convert a tensor to R
- Perform a Boolean operation that is not directly available in PyTorch.

2.10 Convert a `numpy` object to an `R` object

This is mainly required for these reasons:

1. Create a data structure in R
2. Plot using r-base or ggplot2
3. Perform an analysis on parts of a tensor
4. Use R statistical functions that are not available in PyTorch.

Part II

Basic Tensor Operations

Chapter 3

Tensors

We describe the most important PyTorch methods in this chapter.

3.1 Arithmetic of tensors

3.2 Boolean operations

3.3 Slicing

3.4 Example

The following example was converted from PyTorch to rTorch to show differences and similarities of both approaches. The original source can be found here:

Source: https://github.com/jcjohnson/pytorch-examples#pytorch-tensors

3.4.1 Load the libraries

```
library(rTorch)

device = torch$device('cpu')
# device = torch.device('cuda') # Uncomment this to run on GPU

torch$manual_seed(0)
#> <torch._C.Generator>
```

- `N` is batch size;
- `D_in` is input dimension;
- `H` is hidden dimension;
- `D_out` is output dimension.

3.4.2 Datasets

We will create a random dataset for a two layer neural network.

```
N <- 64L; D_in <- 1000L; H <- 100L; D_out <- 10L

# Create random Tensors to hold inputs and outputs
x <- torch$randn(N, D_in, device=device)
y <- torch$randn(N, D_out, device=device)
```

```
# Randomly initialize weights
w1 <- torch$randn(D_in, H, device=device)   # layer 1
w2 <- torch$randn(H, D_out, device=device)  # layer 2
```

3.4.3 Run the model

```
learning_rate = 1e-6

# loop
for (t in 1:50) {
  # Forward pass: compute predicted y
  h <- x$mm(w1)
  h_relu <- h$clamp(min=0)
  y_pred <- h_relu$mm(w2)

  # Compute and print loss; loss is a scalar, and is stored in a PyTorch Tensor
  # of shape (); we can get its value as a Python number with loss.item().
  loss <- (torch$sub(y_pred, y))$pow(2)$sum()
  cat(t, "\t")
  cat(loss$item(), "\n")

  # Backprop to compute gradients of w1 and w2 with respect to loss
  grad_y_pred <- torch$mul(torch$scalar_tensor(2.0), torch$sub(y_pred, y))
  grad_w2 <- h_relu$t()$mm(grad_y_pred)
  grad_h_relu <- grad_y_pred$mm(w2$t())
  grad_h <- grad_h_relu$clone()
  # grad_h[h < 0] = 0
  mask <- grad_h$lt(0)
  # print(mask)
```

```
  # negatives <- torch$masked_select(grad_h, mask)
  # print(negatives)
  # negatives <- 0.0

  torch$masked_select(grad_h, mask)$fill_(0.0)

  # print(grad_h)
  grad_w1 <- x$t()$mm(grad_h)

  # Update weights using gradient descent
  w1 <- torch$sub(w1, torch$mul(learning_rate, grad_w1))
  w2 <- torch$sub(w2, torch$mul(learning_rate, grad_w2))
}
#> 1    29428666
#> 2    22572578
#> 3    20474034
#> 4    19486618
#> 5    1.8e+07
#> 6    15345387
#> 7    1.2e+07
#> 8    8557820
#> 9    5777508
#> 10   3791835
#> 11   2494379
#> 12   1679618
#> 13   1176170
#> 14   858874
#> 15   654740
#> 16   517359
#> 17   421628
#> 18   351479
#> 19   298321
#> 20   256309
#> 21   222513
#> 22   194530
#> 23   171048
#> 24   151092
#> 25   134001
#> 26   119256
#> 27   106431
#> 28   95220
#> 29   85393
#> 30   76739
#> 31   69099
#> 32   62340
```

```
#> 33   56344
#> 34   51009
#> 35   46249
#> 36   41992
#> 37   38182
#> 38   34770
#> 39   31705
#> 40   28946
#> 41   26458
#> 42   24211
#> 43   22179
#> 44   20334
#> 45   18659
#> 46   17138
#> 47   15753
#> 48   14494
#> 49   13347
#> 50   12301
```

Chapter 4

Linear Algebra with Torch

The following are basic operations of Linear Algebra using PyTorch.

```
library(rTorch)
```

4.1 Scalars

```
torch$scalar_tensor(2.78654)
#> tensor(2.7865)

torch$scalar_tensor(0L)
#> tensor(0.)

torch$scalar_tensor(1L)
#> tensor(1.)

torch$scalar_tensor(TRUE)
#> tensor(1.)

torch$scalar_tensor(FALSE)
#> tensor(0.)
```

4.2 Vectors

```
v <- c(0, 1, 2, 3, 4, 5)
torch$as_tensor(v)
```

```
#> tensor([0., 1., 2., 3., 4., 5.])
```

```
# row-vector
message("R matrix")
#> R matrix
(mr <- matrix(1:10, nrow=1))
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> [1,]    1    2    3    4    5    6    7    8    9    10
message("as_tensor")
#> as_tensor
torch$as_tensor(mr)
#> tensor([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]], dtype=torch.int32)
message("shape_of_tensor")
#> shape_of_tensor
torch$as_tensor(mr)$shape
#> torch.Size([1, 10])
```

```
# column-vector
message("R matrix, one column")
#> R matrix, one column
(mc <- matrix(1:10, ncol=1))
#>       [,1]
#>  [1,]    1
#>  [2,]    2
#>  [3,]    3
#>  [4,]    4
#>  [5,]    5
#>  [6,]    6
#>  [7,]    7
#>  [8,]    8
#>  [9,]    9
#> [10,]   10
message("as_tensor")
#> as_tensor
torch$as_tensor(mc)
#> tensor([[ 1],
#>         [ 2],
#>         [ 3],
#>         [ 4],
#>         [ 5],
#>         [ 6],
#>         [ 7],
#>         [ 8],
#>         [ 9],
#>         [10]], dtype=torch.int32)
message("size of tensor")
```

```
#> size of tensor
torch$as_tensor(mc)$shape
#> torch.Size([10, 1])
```

4.3 Matrices

```
message("R matrix")
#> R matrix
(m1 <- matrix(1:24, nrow = 3, byrow = TRUE))
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
#> [1,]    1    2    3    4    5    6    7    8
#> [2,]    9   10   11   12   13   14   15   16
#> [3,]   17   18   19   20   21   22   23   24
message("as_tensor")
#> as_tensor
(t1 <- torch$as_tensor(m1))
#> tensor([[ 1,  2,  3,  4,  5,  6,  7,  8],
#>         [ 9, 10, 11, 12, 13, 14, 15, 16],
#>         [17, 18, 19, 20, 21, 22, 23, 24]], dtype=torch.int32)
message("shape")
#> shape
torch$as_tensor(m1)$shape
#> torch.Size([3, 8])
message("size")
#> size
torch$as_tensor(m1)$size()
#> torch.Size([3, 8])
message("dim")
#> dim
dim(torch$as_tensor(m1))
#> [1] 3 8
message("length")
#> length
length(torch$as_tensor(m1))
#> [1] 24
```

```
message("R matrix")
#> R matrix
(m2 <- matrix(0:99, ncol = 10))
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#>  [1,]    0   10   20   30   40   50   60   70   80    90
#>  [2,]    1   11   21   31   41   51   61   71   81    91
#>  [3,]    2   12   22   32   42   52   62   72   82    92
```

```
#>  [4,]    3   13   23   33   43   53   63   73   83    93
#>  [5,]    4   14   24   34   44   54   64   74   84    94
#>  [6,]    5   15   25   35   45   55   65   75   85    95
#>  [7,]    6   16   26   36   46   56   66   76   86    96
#>  [8,]    7   17   27   37   47   57   67   77   87    97
#>  [9,]    8   18   28   38   48   58   68   78   88    98
#> [10,]    9   19   29   39   49   59   69   79   89    99
message("as_tensor")
#> as_tensor
(t2 <- torch$as_tensor(m2))
#> tensor([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
#>         [ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91],
#>         [ 2, 12, 22, 32, 42, 52, 62, 72, 82, 92],
#>         [ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93],
#>         [ 4, 14, 24, 34, 44, 54, 64, 74, 84, 94],
#>         [ 5, 15, 25, 35, 45, 55, 65, 75, 85, 95],
#>         [ 6, 16, 26, 36, 46, 56, 66, 76, 86, 96],
#>         [ 7, 17, 27, 37, 47, 57, 67, 77, 87, 97],
#>         [ 8, 18, 28, 38, 48, 58, 68, 78, 88, 98],
#>         [ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99]], dtype=torch.int32)
message("shape")
#> shape
t2$shape
#> torch.Size([10, 10])
message("dim")
#> dim
dim(torch$as_tensor(m2))
#> [1] 10 10
```

```
m1[1, 1]
#> [1] 1
m2[1, 1]
#> [1] 0
```

```
t1[1, 1]
#> tensor(1, dtype=torch.int32)
t2[1, 1]
#> tensor(0, dtype=torch.int32)
```

4.4 3D+ tensors

```
# RGB color image has three axes
(img <- torch$rand(3L, 28L, 28L))
```

```
#> tensor([[[0.7845, 0.7715, 0.2301,  ..., 0.4217, 0.6940, 0.3484],
#>          [0.1873, 0.8930, 0.4096,  ..., 0.3427, 0.6798, 0.2500],
#>          [0.6450, 0.4386, 0.9824,  ..., 0.7410, 0.4803, 0.9586],
#>          ...,
#>          [0.5346, 0.9148, 0.2918,  ..., 0.0985, 0.7306, 0.3009],
#>          [0.4837, 0.7250, 0.3804,  ..., 0.1770, 0.5224, 0.6434],
#>          [0.0566, 0.9392, 0.1259,  ..., 0.1219, 0.5850, 0.4021]],
#>
#>         [[0.6517, 0.1734, 0.4216,  ..., 0.8640, 0.6105, 0.7382],
#>          [0.5534, 0.9781, 0.5290,  ..., 0.0500, 0.2172, 0.3618],
#>          [0.8515, 0.1031, 0.4550,  ..., 0.6318, 0.3360, 0.6666],
#>          ...,
#>          [0.8404, 0.4600, 0.9418,  ..., 0.0392, 0.0359, 0.6936],
#>          [0.9024, 0.6239, 0.3953,  ..., 0.3193, 0.8596, 0.5661],
#>          [0.7558, 0.2146, 0.3557,  ..., 0.8097, 0.8997, 0.9328]],
#>
#>         [[0.6283, 0.4728, 0.4200,  ..., 0.7974, 0.1109, 0.8935],
#>          [0.4471, 0.7890, 0.6933,  ..., 0.6888, 0.2614, 0.0526],
#>          [0.4849, 0.8989, 0.6823,  ..., 0.1800, 0.4838, 0.3883],
#>          ...,
#>          [0.6025, 0.0733, 0.0385,  ..., 0.7322, 0.7834, 0.0417],
#>          [0.3407, 0.5911, 0.5052,  ..., 0.1466, 0.2441, 0.7902],
#>          [0.1311, 0.2139, 0.3928,  ..., 0.5024, 0.5955, 0.7353]]])
img$shape
#> torch.Size([3, 28, 28])
```

```
img[1, 1, 1]
#> tensor(0.7845)
img[3, 28, 28]
#> tensor(0.7353)
```

4.5 Transpose of a matrix

```
(m3 <- matrix(1:25, ncol = 5))
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    6   11   16   21
#> [2,]    2    7   12   17   22
#> [3,]    3    8   13   18   23
#> [4,]    4    9   14   19   24
#> [5,]    5   10   15   20   25

# transpose
message("transpose")
```

```
#> transpose
tm3 <- t(m3)
tm3
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,]    1    2    3    4    5
#> [2,]    6    7    8    9   10
#> [3,]   11   12   13   14   15
#> [4,]   16   17   18   19   20
#> [5,]   21   22   23   24   25
```

```
message("as_tensor")
#> as_tensor
(t3 <- torch$as_tensor(m3))
#> tensor([[ 1,  6, 11, 16, 21],
#>         [ 2,  7, 12, 17, 22],
#>         [ 3,  8, 13, 18, 23],
#>         [ 4,  9, 14, 19, 24],
#>         [ 5, 10, 15, 20, 25]], dtype=torch.int32)
message("transpose")
#> transpose
tt3 <- t3$transpose(dim0 = 0L, dim1 = 1L)
tt3
#> tensor([[ 1,  2,  3,  4,  5],
#>         [ 6,  7,  8,  9, 10],
#>         [11, 12, 13, 14, 15],
#>         [16, 17, 18, 19, 20],
#>         [21, 22, 23, 24, 25]], dtype=torch.int32)
```

```
tm3 == tt3$numpy()   # convert first the tensor to numpy
#>      [,1] [,2] [,3] [,4] [,5]
#> [1,] TRUE TRUE TRUE TRUE TRUE
#> [2,] TRUE TRUE TRUE TRUE TRUE
#> [3,] TRUE TRUE TRUE TRUE TRUE
#> [4,] TRUE TRUE TRUE TRUE TRUE
#> [5,] TRUE TRUE TRUE TRUE TRUE
```

4.6 Vectors, special case of a matrix

```
message("R matrix")
#> R matrix
m2 <- matrix(0:99, ncol = 10)
message("as_tensor")
#> as_tensor
```

```
(t2 <- torch$as_tensor(m2))
#> tensor([[ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90],
#>         [ 1, 11, 21, 31, 41, 51, 61, 71, 81, 91],
#>         [ 2, 12, 22, 32, 42, 52, 62, 72, 82, 92],
#>         [ 3, 13, 23, 33, 43, 53, 63, 73, 83, 93],
#>         [ 4, 14, 24, 34, 44, 54, 64, 74, 84, 94],
#>         [ 5, 15, 25, 35, 45, 55, 65, 75, 85, 95],
#>         [ 6, 16, 26, 36, 46, 56, 66, 76, 86, 96],
#>         [ 7, 17, 27, 37, 47, 57, 67, 77, 87, 97],
#>         [ 8, 18, 28, 38, 48, 58, 68, 78, 88, 98],
#>         [ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99]], dtype=torch.int32)

# in R
message("select column of matrix")
#> select column of matrix
(v1 <- m2[, 1])
#>  [1] 0 1 2 3 4 5 6 7 8 9
message("select row of matrix")
#> select row of matrix
(v2 <- m2[10, ])
#>  [1]  9 19 29 39 49 59 69 79 89 99
```

```
# PyTorch
message()
#>
t2c <- t2[, 1]
t2r <- t2[10, ]

t2c
#> tensor([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=torch.int32)
t2r
#> tensor([ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99], dtype=torch.int32)
```

In vectors, the vector and its transpose are equal.

```
tt2r <- t2r$transpose(dim0 = 0L, dim1 = 0L)
tt2r
#> tensor([ 9, 19, 29, 39, 49, 59, 69, 79, 89, 99], dtype=torch.int32)
```

```
# a tensor of booleans. is vector equal to its transposed?
t2r == tt2r
#> tensor([True, True, True, True, True, True, True, True, True, True],
#>        dtype=torch.bool)
```

4.7 Tensor arithmetic

```
message("x")
#> x
(x = torch$ones(5L, 4L))
#> tensor([[1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.]])
message("y")
#> y
(y = torch$ones(5L, 4L))
#> tensor([[1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.]])
message("x+y")
#> x+y
x + y
#> tensor([[2., 2., 2., 2.],
#>         [2., 2., 2., 2.],
#>         [2., 2., 2., 2.],
#>         [2., 2., 2., 2.],
#>         [2., 2., 2., 2.]])
```

$$A + B = B + A$$

```
x + y == y + x
#> tensor([[True, True, True, True],
#>         [True, True, True, True],
#>         [True, True, True, True],
#>         [True, True, True, True],
#>         [True, True, True, True]], dtype=torch.bool)
```

4.8 Add a scalar to a tensor

```
s <- 0.5    # scalar
x + s
#> tensor([[1.5000, 1.5000, 1.5000, 1.5000],
#>         [1.5000, 1.5000, 1.5000, 1.5000],
```

```
#>          [1.5000, 1.5000, 1.5000, 1.5000],
#>          [1.5000, 1.5000, 1.5000, 1.5000],
#>          [1.5000, 1.5000, 1.5000, 1.5000]])
```

```
# scalar multiplying two tensors
s * (x + y)
#> tensor([[1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.]])
```

4.9 Multiplying tensors

$$A * B = B * A$$

```
message("x")
#> x
(x = torch$ones(5L, 4L))
#> tensor([[1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.]])
message("y")
#> y
(y = torch$ones(5L, 4L))
#> tensor([[1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.],
#>         [1., 1., 1., 1.]])
message("2x+4y")
#> 2x+4y
(z = 2 * x + 4 * y)
#> tensor([[6., 6., 6., 6.],
#>         [6., 6., 6., 6.],
#>         [6., 6., 6., 6.],
#>         [6., 6., 6., 6.],
#>         [6., 6., 6., 6.]])
```

```
x * y == y * x
#> tensor([[True, True, True, True],
#>         [True, True, True, True],
```

```
#>          [True, True, True, True],
#>          [True, True, True, True],
#>          [True, True, True, True]], dtype=torch.bool)
```

4.10 Dot product

$$dot(a, b)_{i,j,k,a,b,c} = \sum_m a_{i,j,k,m} b_{a,b,m,c}$$

```
torch$dot(torch$tensor(c(2, 3)), torch$tensor(c(2, 1)))
#> tensor(7.)
```

```
a <- np$array(list(list(1, 2), list(3, 4)))
a
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4
b <- np$array(list(list(1, 2), list(3, 4)))
b
#>      [,1] [,2]
#> [1,]    1    2
#> [2,]    3    4

np$dot(a, b)
#>      [,1] [,2]
#> [1,]    7   10
#> [2,]   15   22
```

`torch.dot()` treats both a and b as 1D vectors (irrespective of their original shape) and computes their inner product.

```
at <- torch$as_tensor(a)
bt <- torch$as_tensor(b)

# torch$dot(at, bt)  <- RuntimeError: dot: Expected 1-D argument self, but got 2-D
# at %.*% bt
```

If we perform the same dot product operation in Python, we get the same error:

```
import torch
import numpy as np

a = np.array([[1, 2], [3, 4]])
a
#> array([[1, 2],
```

```
#>        [3, 4]])
b = np.array([[1, 2], [3, 4]])
b
#> array([[1, 2],
#>        [3, 4]])
np.dot(a, b)
#> array([[ 7, 10],
#>        [15, 22]])
at = torch.as_tensor(a)
bt = torch.as_tensor(b)

at
#> tensor([[1, 2],
#>         [3, 4]])
bt
#> tensor([[1, 2],
#>         [3, 4]])
torch.dot(at, bt)
#> Error in py_call_impl(callable, dots$args, dots$keywords): RuntimeError: dot: Expected 1-D arg
#>
#> Detailed traceback:
#>   File "<string>", line 1, in <module>
```

```
a <- torch$Tensor(list(list(1, 2), list(3, 4)))
b <- torch$Tensor(c(c(1, 2), c(3, 4)))
c <- torch$Tensor(list(list(11, 12), list(13, 14)))

a
#> tensor([[1., 2.],
#>         [3., 4.]])
b
#> tensor([1., 2., 3., 4.])
torch$dot(a, b)
#> Error in py_call_impl(callable, dots$args, dots$keywords): RuntimeError: dot: Expected 1-D arg

# this is another way of performing dot product in PyTorch
# a$dot(a)
```

```
o1 <- torch$ones(2L, 2L)
o2 <- torch$ones(2L, 2L)

o1
#> tensor([[1., 1.],
#>         [1., 1.]])
o2
#> tensor([[1., 1.],
```

```
#>         [1., 1.]])

torch$dot(o1, o2)
#> Error in py_call_impl(callable, dots$args, dots$keywords): RuntimeError: dot: Expec
o1$dot(o2)
#> Error in py_call_impl(callable, dots$args, dots$keywords): RuntimeError: dot: Expec
```

```
# 1D tensors work fine
r = torch$dot(torch$Tensor(list(4L, 2L, 4L)), torch$Tensor(list(3L, 4L, 1L)))
r
#> tensor(24.)
```

```
## mm and matmul seem to address the dot product we are looking for in tensors
a = torch$randn(2L, 3L)
b = torch$randn(3L, 4L)

a$mm(b)
#> tensor([[ 0.0995, -0.5739, -0.9001,  1.2623],
#>         [ 2.7524, -0.0434, -0.6572,  2.8573]])
a$matmul(b)
#> tensor([[ 0.0995, -0.5739, -0.9001,  1.2623],
#>         [ 2.7524, -0.0434, -0.6572,  2.8573]])
```

Here is agood explanation: https://stackoverflow.com/a/44525687/5270873

```
abt <- torch$mm(a, b)$transpose(dim0=0L, dim1=1L)
abt
#> tensor([[ 0.0995,  2.7524],
#>         [-0.5739, -0.0434],
#>         [-0.9001, -0.6572],
#>         [ 1.2623,  2.8573]])
```

```
at <- a$transpose(dim0=0L, dim1=1L)
bt <- b$transpose(dim0=0L, dim1=1L)

btat <- torch$matmul(bt, at)
btat
#> tensor([[ 0.0995,  2.7524],
#>         [-0.5739, -0.0434],
#>         [-0.9001, -0.6572],
#>         [ 1.2623,  2.8573]])
```

$$(AB)^T = B^T A^T$$

```
# tolerance
torch$allclose(abt, btat, rtol=0.0001)
```

```
#> [1] TRUE
```

Part III

Logistic Regression

Chapter 5

Example 1: MNIST handwritten digits

Source: https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/01-basics/logistic_regression/main.py

```
library(rTorch)

nn          <- torch$nn
transforms  <- torchvision$transforms

torch$set_default_dtype(torch$float)
```

5.1 Hyperparameters

```
# Hyper-parameters
input_size    <- 784L
num_classes   <- 10L
num_epochs    <- 5L
batch_size    <- 100L
learning_rate <- 0.001
```

5.2 Read datasets

```
# MNIST dataset (images and labels)
# IDX format
```

```
local_folder <- '../datasets/raw_data'
train_dataset = torchvision$datasets$MNIST(root=local_folder,
                                           train=TRUE,
                                           transform=transforms$ToTensor(),
                                           download=TRUE)

test_dataset = torchvision$datasets$MNIST(root=local_folder,
                                          train=FALSE,
                                          transform=transforms$ToTensor())

# Data loader (input pipeline)
train_loader = torch$utils$data$DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=TRUE)

test_loader = torch$utils$data$DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=FALSE)
```

```
class(train_loader)
#> [1] "torch.utils.data.dataloader.DataLoader"
#> [2] "python.builtin.object"
length(train_loader)
#> [1] 2
```

5.3 Define the model

```
# Logistic regression model
model = nn$Linear(input_size, num_classes)

# Loss and optimizer
# nn.CrossEntropyLoss() computes softmax internally
criterion = nn$CrossEntropyLoss()
optimizer = torch$optim$SGD(model$parameters(), lr=learning_rate)
print(model)
#> Linear(in_features=784, out_features=10, bias=True)
```

5.4 Training

```
# Train the model
iter_train_loader <- iterate(train_loader)
```

```
total_step <-length(iter_train_loader)
```

```
for (epoch in 1:num_epochs) {
    i <-  0
    for (obj in iter_train_loader) {

        images <- obj[[1]]   # tensor torch.Size([64, 3, 28, 28])
        labels <- obj[[2]]   # tensor torch.Size([64]), labels from 0 to 9
        # cat(i, "\t"); print(images$shape)

        # Reshape images to (batch_size, input_size)
        images <- images$reshape(-1L, 28L*28L)
        # images <- torch$as_tensor(images$reshape(-1L, 28L*28L), dtype=torch$double)

        # Forward pass
        outputs <- model(images)
        loss <- criterion(outputs, labels)

        # Backward and optimize
        optimizer$zero_grad()
        loss$backward()
        optimizer$step()

        if ((i+1) %% 100 == 0) {
            cat(sprintf('Epoch [%d/%d], Step [%d/%d], Loss: %f \n',
                epoch+1, num_epochs, i+1, total_step, loss$item()))
        }
        i <-  i + 1
    }
}
#> Epoch [2/5], Step [100/600], Loss: 2.228390
#> Epoch [2/5], Step [200/600], Loss: 2.116164
#> Epoch [2/5], Step [300/600], Loss: 2.019095
#> Epoch [2/5], Step [400/600], Loss: 1.994817
#> Epoch [2/5], Step [500/600], Loss: 1.876899
#> Epoch [2/5], Step [600/600], Loss: 1.840720
#> Epoch [3/5], Step [100/600], Loss: 1.690704
#> Epoch [3/5], Step [200/600], Loss: 1.681273
#> Epoch [3/5], Step [300/600], Loss: 1.619535
#> Epoch [3/5], Step [400/600], Loss: 1.650310
#> Epoch [3/5], Step [500/600], Loss: 1.541979
#> Epoch [3/5], Step [600/600], Loss: 1.520510
#> Epoch [4/5], Step [100/600], Loss: 1.362663
#> Epoch [4/5], Step [200/600], Loss: 1.406761
#> Epoch [4/5], Step [300/600], Loss: 1.364937
```

```
#> Epoch [4/5], Step [400/600], Loss: 1.420470
#> Epoch [4/5], Step [500/600], Loss: 1.323673
#> Epoch [4/5], Step [600/600], Loss: 1.311175
#> Epoch [5/5], Step [100/600], Loss: 1.151503
#> Epoch [5/5], Step [200/600], Loss: 1.225978
#> Epoch [5/5], Step [300/600], Loss: 1.196704
#> Epoch [5/5], Step [400/600], Loss: 1.259773
#> Epoch [5/5], Step [500/600], Loss: 1.173820
#> Epoch [5/5], Step [600/600], Loss: 1.167180
#> Epoch [6/5], Step [100/600], Loss: 1.008452
#> Epoch [6/5], Step [200/600], Loss: 1.100466
#> Epoch [6/5], Step [300/600], Loss: 1.079904
#> Epoch [6/5], Step [400/600], Loss: 1.142588
#> Epoch [6/5], Step [500/600], Loss: 1.065565
#> Epoch [6/5], Step [600/600], Loss: 1.062986
```

5.5 Prediction

```
# Adjust weights and reset gradients
iter_test_loader <- iterate(test_loader)

with(torch$no_grad(), {
    correct <-  0
    total <-  0
    for (obj in iter_test_loader) {
        images <- obj[[1]]   # tensor torch.Size([64, 3, 28, 28])
        labels <- obj[[2]]   # tensor torch.Size([64]), labels from 0 to 9
        images = images$reshape(-1L, 28L*28L)
        # images <- torch$as_tensor(images$reshape(-1L, 28L*28L), dtype=torch$double)
        outputs = model(images)
        .predicted = torch$max(outputs$data, 1L)
        predicted <- .predicted[1L]
        total = total + labels$size(0L)
        correct = correct + sum((predicted$numpy() == labels$numpy()))
    }
    cat(sprintf('Accuracy of the model on the 10000 test images: %f %%', (100 * correct

})
#> Accuracy of the model on the 10000 test images: 82.830000 %
```

5.6 Save the model

```
# Save the model checkpoint
torch$save(model$state_dict(), 'model.ckpt')
```

Part IV

Linear Regression

Chapter 6

Simple linear regression

6.1 Introduction

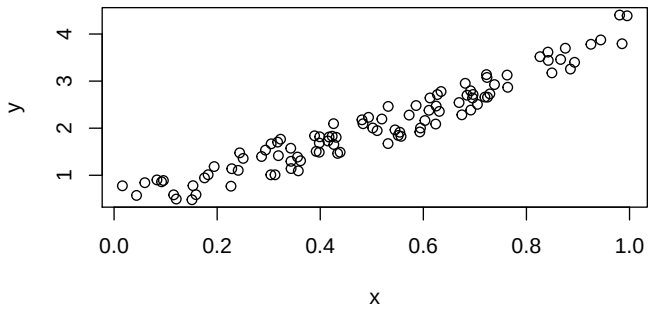
Source: https://www.guru99.com/pytorch-tutorial.html

```
library(rTorch)

nn       <- torch$nn
Variable <- torch$autograd$Variable

torch$manual_seed(123)
#> <torch._C.Generator>
```

6.2 Generate the dataset

Before you start the training process, you need to know our data. You make a random function to test our model. $Y = x3sin(x) + 3x + 0.8rand(100)$

```
np$random$seed(123L)

x = np$random$rand(100L)
y = np$sin(x) * np$power(x, 3L) + 3L * x + np$random$rand(100L) * 0.8

plot(x, y)
```



6.3 Convert arrays to tensors

Before you start the training process, you need to convert the numpy array to Variables that supported by Torch and autograd.

6.4 Converting from numpy to tensor

Notice that before converting to a Torch tensor, we need first to convert the R numeric vector to a `numpy` array:

```
# convert numpy array to tensor in shape of input size
x <- r_to_py(x)
y <- r_to_py(y)
x = torch$from_numpy(x$reshape(-1L, 1L))$float()
y = torch$from_numpy(y$reshape(-1L, 1L))$float()
print(x, y)
#> tensor([[0.6965],
#>         [0.2861],
#>         [0.2269],
#>         [0.5513],
#>         [0.7195],
#>         [0.4231],
#>         [0.9808],
#>         [0.6848],
#>         [0.4809],
#>         [0.3921],
#>         [0.3432],
#>         [0.7290],
#>         [0.4386],
#>         [0.0597],
#>         [0.3980],
#>         [0.7380],
```

```
#>          [0.1825],
#>          [0.1755],
#>          [0.5316],
#>          [0.5318],
#>          [0.6344],
#>          [0.8494],
#>          [0.7245],
#>          [0.6110],
#>          [0.7224],
#>          [0.3230],
#>          [0.3618],
#>          [0.2283],
#>          [0.2937],
#>          [0.6310],
#>          [0.0921],
#>          [0.4337],
#>          [0.4309],
#>          [0.4937],
#>          [0.4258],
#>          [0.3123],
#>          [0.4264],
#>          [0.8934],
#>          [0.9442],
#>          [0.5018],
#>          [0.6240],
#>          [0.1156],
#>          [0.3173],
#>          [0.4148],
#>          [0.8663],
#>          [0.2505],
#>          [0.4830],
#>          [0.9856],
#>          [0.5195],
#>          [0.6129],
#>          [0.1206],
#>          [0.8263],
#>          [0.6031],
#>          [0.5451],
#>          [0.3428],
#>          [0.3041],
#>          [0.4170],
#>          [0.6813],
#>          [0.8755],
#>          [0.5104],
#>          [0.6693],
```

```
#>         [0.5859],
#>         [0.6249],
#>         [0.6747],
#>         [0.8423],
#>         [0.0832],
#>         [0.7637],
#>         [0.2437],
#>         [0.1942],
#>         [0.5725],
#>         [0.0957],
#>         [0.8853],
#>         [0.6272],
#>         [0.7234],
#>         [0.0161],
#>         [0.5944],
#>         [0.5568],
#>         [0.1590],
#>         [0.1531],
#>         [0.6955],
#>         [0.3188],
#>         [0.6920],
#>         [0.5544],
#>         [0.3890],
#>         [0.9251],
#>         [0.8417],
#>         [0.3574],
#>         [0.0436],
#>         [0.3048],
#>         [0.3982],
#>         [0.7050],
#>         [0.9954],
#>         [0.3559],
#>         [0.7625],
#>         [0.5932],
#>         [0.6917],
#>         [0.1511],
#>         [0.3989],
#>         [0.2409],
#>         [0.3435]])
```

6.5 Creating the network model

Our network model is a simple Linear layer with an input and an output shape of one.

And the network output should be like this

```
Net(
  (hidden): Linear(in_features=1, out_features=1, bias=True)
)
```

```
py_run_string("import torch")
main = py_run_string(
"
import torch.nn as nn

class Net(nn.Module):
   def __init__(self):
       super(Net, self).__init__()
       self.layer = torch.nn.Linear(1, 1)

   def forward(self, x):
       x = self.layer(x)
       return x
")


# build a Linear Rgression model
net <- main$Net()

print(net)
#> Net(
#>   (layer): Linear(in_features=1, out_features=1, bias=True)
#> )
```

6.6 Optimizer and Loss

Next, you should define the Optimizer and the Loss Function for our training process.

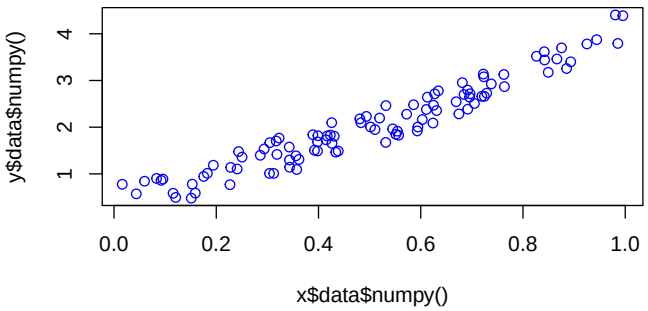
```
# Define Optimizer and Loss Function
optimizer <- torch$optim$SGD(net$parameters(), lr=0.2)
loss_func <- torch$nn$MSELoss()
print(optimizer)
#> SGD (
```

```
#> Parameter Group 0
#>     dampening: 0
#>     lr: 0.2
#>     momentum: 0
#>     nesterov: False
#>     weight_decay: 0
#> )
print(loss_func)
#> MSELoss()
```

6.7 Training

Now let's start our training process. With an epoch of 250, you will iterate our data to find the best value for our hyperparameters.

```
# x = x$type(torch$float)   # make it a a FloatTensor
# y = y$type(torch$float)

# x <- torch$as_tensor(x, dtype = torch$float)
# y <- torch$as_tensor(y, dtype = torch$float)

inputs  = Variable(x)
outputs = Variable(y)

# base plot
plot(x$data$numpy(), y$data$numpy(), col = "blue")
for (i in 1:250) {
   prediction = net(inputs)
   loss = loss_func(prediction, outputs)
   optimizer$zero_grad()
   loss$backward()
   optimizer$step()

   if (i > 1) break

   if (i %% 10 == 0) {
       # plot and show learning process
      # points(x$data$numpy(), y$data$numpy())
      points(x$data$numpy(), prediction$data$numpy(), col="red")
       # cat(i, loss$data$numpy(), "\n")
   }
}
```



6.8 Results

As you can see below, you successfully performed regression with a neural network. Actually, on every iteration, the red line in the plot will update and change its position to fit the data. But in this picture, you only show you the final result.

Chapter 7

Rainfall. Linear Regression

```
library(rTorch)
```

Select the device: CPU or GPU

```
torch$manual_seed(0)
#> <torch._C.Generator>

device = torch$device('cpu')
```

7.1 Training data

The training data can be represented using 2 matrices (inputs and targets), each with one row per observation, and one column per variable.

```
# Input (temp, rainfall, humidity)
inputs = np$array(list(list(73, 67, 43),
                   list(91, 88, 64),
                   list(87, 134, 58),
                   list(102, 43, 37),
                   list(69, 96, 70)), dtype='float32')

# Targets (apples, oranges)
targets = np$array(list(list(56, 70),
                    list(81, 101),
                    list(119, 133),
                    list(22, 37),
                    list(103, 119)), dtype='float32')
```

7.2 Convert arrays to tensors

Before we build a model, we need to convert inputs and targets to PyTorch tensors.

```
# Convert inputs and targets to tensors
inputs = torch$from_numpy(inputs)
targets = torch$from_numpy(targets)

print(inputs)
#> tensor([[ 73.,  67.,  43.],
#>         [ 91.,  88.,  64.],
#>         [ 87., 134.,  58.],
#>         [102.,  43.,  37.],
#>         [ 69.,  96.,  70.]], dtype=torch.float64)
print(targets)
#> tensor([[ 56.,  70.],
#>         [ 81., 101.],
#>         [119., 133.],
#>         [ 22.,  37.],
#>         [103., 119.]], dtype=torch.float64)
```

The weights and biases can also be represented as matrices, initialized with random values. The first row of w and the first element of b are used to predict the first target variable, i.e. yield for apples, and, similarly, the second for oranges.

```
# random numbers for weights and biases. Then convert to double()
torch$set_default_dtype(torch$double)

w = torch$randn(2L, 3L, requires_grad=TRUE)  #$double()
b = torch$randn(2L, requires_grad=TRUE)      #$double()

print(w)
#> tensor([[ 1.5410, -0.2934, -2.1788],
#>         [ 0.5684, -1.0845, -1.3986]], requires_grad=True)
print(b)
#> tensor([0.4033, 0.8380], requires_grad=True)
```

7.3 Build the model

The model is simply a function that performs a matrix multiplication of the input x and the weights w (transposed), and adds the bias b (replicated for each observation).

```
model <- function(x) {
  wt <- w$t()
  return(torch$add(torch$mm(x, wt), b))
}
```

7.4 Generate predictions

The matrix obtained by passing the input data to the model is a set of predictions for the target variables.

```
# Generate predictions
preds = model(inputs)
print(preds)
#> tensor([[  -0.4516,  -90.4691],
#>         [ -24.6303, -132.3828],
#>         [ -31.2192, -176.1530],
#>         [  64.3523,  -39.5645],
#>         [ -73.9524, -161.9560]], grad_fn=<AddBackward0>)
```

```
# Compare with targets
print(targets)
#> tensor([[ 56.,  70.],
#>         [ 81., 101.],
#>         [119., 133.],
#>         [ 22.,  37.],
#>         [103., 119.]])
```

Because we've started with random weights and biases, the model does not a very good job of predicting the target variables.

7.5 Loss Function

We can compare the predictions with the actual targets, using the following method:

- Calculate the difference between the two matrices (preds and targets).
- Square all elements of the difference matrix to remove negative values.
- Calculate the average of the elements in the resulting matrix.

The result is a single number, known as the mean squared error (MSE).

```
# MSE loss
mse = function(t1, t2) {
  diff <- torch$sub(t1, t2)
```

```
  mul <- torch$sum(torch$mul(diff, diff))
  return(torch$div(mul, diff$numel()))
}

print(mse)
#> function(t1, t2) {
#>   diff <- torch$sub(t1, t2)
#>   mul <- torch$sum(torch$mul(diff, diff))
#>   return(torch$div(mul, diff$numel()))
#> }
```

7.6 Step by step process

7.6.1 Compute the losses

```
# Compute loss
loss = mse(preds, targets)
print(loss)
#> tensor(33060.8053, grad_fn=<DivBackward0>)
# 46194
# 33060.8070
```

The resulting number is called the **loss**, because it indicates how bad the model is at predicting the target variables. Lower the loss, better the model.

7.6.2 Compute Gradients

With PyTorch, we can automatically compute the gradient or derivative of the loss w.r.t. to the weights and biases, because they have `requires_grad` set to True.

```
# Compute gradients
loss$backward()
```

The gradients are stored in the .grad property of the respective tensors.

```
# Gradients for weights
print(w)
#> tensor([[ 1.5410, -0.2934, -2.1788],
#>         [ 0.5684, -1.0845, -1.3986]], requires_grad=True)
print(w$grad)
#> tensor([[ -6938.4351,  -9674.6757,  -5744.0206],
#>         [-17408.7861, -20595.9333, -12453.4702]])
```

```
# Gradients for bias
print(b)
#> tensor([0.4033, 0.8380], requires_grad=True)
print(b$grad)
#> tensor([ -89.3802, -212.1051])
```

A key insight from calculus is that the gradient indicates the rate of change of the loss, or the slope of the loss function w.r.t. the weights and biases.

- If a gradient element is positive:
  - increasing the element's value slightly will increase the loss.
  - decreasing the element's value slightly will decrease the loss.
- If a gradient element is negative,
  - increasing the element's value slightly will decrease the loss.
  - decreasing the element's value slightly will increase the loss.

The increase or decrease is proportional to the value of the gradient.

7.6.3 Reset the gradients

Finally, we'll reset the gradients to zero before moving forward, because PyTorch accumulates gradients.

```
# Reset the gradients
w$grad$zero_()
#> tensor([[0., 0., 0.],
#>         [0., 0., 0.]])
b$grad$zero_()
#> tensor([0., 0.])

print(w$grad)
#> tensor([[0., 0., 0.],
#>         [0., 0., 0.]])
print(b$grad)
#> tensor([0., 0.])
```

7.6.3.1 Adjust weights and biases using gradient descent

We'll reduce the loss and improve our model using the gradient descent algorithm, which has the following steps:

1. Generate predictions
2. Calculate the loss
3. Compute gradients w.r.t the weights and biases
4. Adjust the weights by subtracting a small quantity proportional to the gradient

5. Reset the gradients to zero

```
# Generate predictions
preds = model(inputs)
print(preds)
#> tensor([[  -0.4516,  -90.4691],
#>         [ -24.6303, -132.3828],
#>         [ -31.2192, -176.1530],
#>         [  64.3523,  -39.5645],
#>         [ -73.9524, -161.9560]], grad_fn=<AddBackward0>)
```

```
# Calculate the loss
loss = mse(preds, targets)
print(loss)
#> tensor(33060.8053, grad_fn=<DivBackward0>)
```

```
# Compute gradients
loss$backward()

print(w$grad)
#> tensor([[ -6938.4351,  -9674.6757,  -5744.0206],
#>         [-17408.7861, -20595.9333, -12453.4702]])
print(b$grad)
#> tensor([ -89.3802, -212.1051])
```

```
# Adjust weights and reset gradients
with(torch$no_grad(), {
  print(w); print(b)    # requires_grad attribute remains
  w$data <- torch$sub(w$data, torch$mul(w$grad$data, torch$scalar_tensor(1e-5)))
  b$data <- torch$sub(b$data, torch$mul(b$grad$data, torch$scalar_tensor(1e-5)))

  print(w$grad$data$zero_())
  print(b$grad$data$zero_())
})
#> tensor([[ 1.5410, -0.2934, -2.1788],
#>         [ 0.5684, -1.0845, -1.3986]], requires_grad=True)
#> tensor([0.4033, 0.8380], requires_grad=True)
#> tensor([[0., 0., 0.],
#>         [0., 0., 0.]])
#> tensor([0., 0.])

print(w)
#> tensor([[ 1.6104, -0.1967, -2.1213],
#>         [ 0.7425, -0.8786, -1.2741]], requires_grad=True)
print(b)
#> tensor([0.4042, 0.8401], requires_grad=True)
```

With the new weights and biases, the model should have a lower loss.

```
# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
print(loss)
#> tensor(23432.4894, grad_fn=<DivBackward0>)
```

7.7 All together: train for multiple epochs

To reduce the loss further, we repeat the process of adjusting the weights and biases using the gradients multiple times. Each iteration is called an **epoch**.

```
# Running all together
# Adjust weights and reset gradients
num_epochs <- 100

for (i in 1:num_epochs) {
  preds = model(inputs)
  loss = mse(preds, targets)
  loss$backward()
  with(torch$no_grad(), {
    w$data <- torch$sub(w$data, torch$mul(w$grad, torch$scalar_tensor(1e-5)))
    b$data <- torch$sub(b$data, torch$mul(b$grad, torch$scalar_tensor(1e-5)))

    w$grad$zero_()
    b$grad$zero_()
  })
}

# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
print(loss)
#> tensor(1258.0216, grad_fn=<DivBackward0>)

# predictions
preds
#> tensor([[ 69.2462,  80.2082],
#>         [ 73.7183,  97.2052],
#>         [118.5780, 124.9272],
#>         [ 89.2282,  92.7052],
#>         [ 47.4648,  80.7782]], grad_fn=<AddBackward0>)

# Targets
```

```
targets
#> tensor([[ 56.,  70.],
#>         [ 81., 101.],
#>         [119., 133.],
#>         [ 22.,  37.],
#>         [103., 119.]])
```

Part V

Neural Networks

Chapter 8

A very simple neural network

8.1 Introduction

Source: https://github.com/jcjohnson/pytorch-examples#pytorch-nn

In this example we use the torch `nn` package to implement our two-layer network:

8.2 Select device

```
library(rTorch)

device = torch$device('cpu')

# device = torch.device('cuda') # Uncomment this to run on GPU
```

- `N` is batch size;
- `D_in` is input dimension;
- `H` is hidden dimension;
- `D_out` is output dimension.

8.3 Create the dataset

```
torch$manual_seed(0)
#> <torch._C.Generator>
```

```
N <- 64L; D_in <- 1000L; H <- 100L; D_out <- 10L

# Create random Tensors to hold inputs and outputs
x = torch$randn(N, D_in, device=device)
y = torch$randn(N, D_out, device=device)
```

8.4 Define the model

Use the `nn` package to define our model as a sequence of layers. `nn.Sequential` is a Module which contains other Modules, and applies them in sequence to produce its output. Each Linear Module computes output from input using a linear function, and holds internal Tensors for its weight and bias. After constructing the model we use the `.to()` method to move it to the desired device.

```
model <- torch$nn$Sequential(
  torch$nn$Linear(D_in, H),              # first layer
  torch$nn$ReLU(),
  torch$nn$Linear(H, D_out))$to(device)  # output layer

print(model)
#> Sequential(
#>   (0): Linear(in_features=1000, out_features=100, bias=True)
#>   (1): ReLU()
#>   (2): Linear(in_features=100, out_features=10, bias=True)
#> )
```

8.5 Loss function

The `nn` package also contains definitions of popular loss functions; in this case we will use Mean Squared Error (**MSE**) as our loss function. Setting `reduction='sum'` means that we are computing the *sum* of squared errors rather than the mean; this is for consistency with the examples above where we manually compute the loss, but in practice it is more common to use mean squared error as a loss by setting `reduction='elementwise_mean'`.

```
loss_fn = torch$nn$MSELoss(reduction = 'sum')
```

8.6 Iterate through batches

```
learning_rate = 1e-4

for (t in 1:500) {
  # Forward pass: compute predicted y by passing x to the model. Module objects
  # override the __call__ operator so you can call them like functions. When
  # doing so you pass a Tensor of input data to the Module and it produces
  # a Tensor of output data.
  y_pred = model(x)

  # Compute and print loss. We pass Tensors containing the predicted and true
  # values of y, and the loss function returns a Tensor containing the loss.
  loss = loss_fn(y_pred, y)

  cat(t, "\t")
  cat(loss$item(), "\n")

  # Zero the gradients before running the backward pass.
  model$zero_grad()

  # Backward pass: compute gradient of the loss with respect to all the learnable
  # parameters of the model. Internally, the parameters of each Module are stored
  # in Tensors with requires_grad=True, so this call will compute gradients for
  # all learnable parameters in the model.
  loss$backward()

  # Update the weights using gradient descent. Each parameter is a Tensor, so
  # we can access its data and gradients like we did before.
  with(torch$no_grad(), {
      for (param in iterate(model$parameters())) {
        # in Python this code is much simpler. In R we have to do some conversions

        # param$data <- torch$sub(param$data,
        #                         torch$mul(param$grad$float(),
        #                           torch$scalar_tensor(learning_rate)))

        param$data <- param$data - param$grad * learning_rate
      }
   })
}
#> 1    628
#> 2    585
#> 3    547
#> 4    513
```

```
#> 5    482
#> 6    455
#> 7    430
#> 8    406
#> 9    385
#> 10   364
#> 11   345
#> 12   328
#> 13   311
#> 14   295
#> 15   280
#> 16   265
#> 17   252
#> 18   239
#> 19   226
#> 20   214
#> 21   203
#> 22   192
#> 23   181
#> 24   172
#> 25   162
#> 26   153
#> 27   145
#> 28   137
#> 29   129
#> 30   122
#> 31   115
#> 32   109
#> 33   103
#> 34   96.9
#> 35   91.5
#> 36   86.3
#> 37   81.5
#> 38   76.9
#> 39   72.6
#> 40   68.5
#> 41   64.6
#> 42   61
#> 43   57.6
#> 44   54.3
#> 45   51.3
#> 46   48.5
#> 47   45.8
#> 48   43.2
#> 49   40.9
```

```
#> 50   38.6
#> 51   36.5
#> 52   34.5
#> 53   32.7
#> 54   30.9
#> 55   29.3
#> 56   27.8
#> 57   26.3
#> 58   24.9
#> 59   23.7
#> 60   22.4
#> 61   21.3
#> 62   20.2
#> 63   19.2
#> 64   18.2
#> 65   17.3
#> 66   16.5
#> 67   15.7
#> 68   14.9
#> 69   14.2
#> 70   13.5
#> 71   12.9
#> 72   12.3
#> 73   11.7
#> 74   11.1
#> 75   10.6
#> 76   10.1
#> 77   9.67
#> 78   9.24
#> 79   8.82
#> 80   8.42
#> 81   8.05
#> 82   7.69
#> 83   7.35
#> 84   7.03
#> 85   6.72
#> 86   6.43
#> 87   6.16
#> 88   5.9
#> 89   5.65
#> 90   5.41
#> 91   5.18
#> 92   4.97
#> 93   4.76
#> 94   4.57
```

```
#> 95   4.38
#> 96   4.2
#> 97   4.03
#> 98   3.87
#> 99   3.72
#> 100  3.57
#> 101  3.43
#> 102  3.29
#> 103  3.17
#> 104  3.04
#> 105  2.92
#> 106  2.81
#> 107  2.7
#> 108  2.6
#> 109  2.5
#> 110  2.41
#> 111  2.31
#> 112  2.23
#> 113  2.14
#> 114  2.06
#> 115  1.99
#> 116  1.91
#> 117  1.84
#> 118  1.77
#> 119  1.71
#> 120  1.65
#> 121  1.59
#> 122  1.53
#> 123  1.47
#> 124  1.42
#> 125  1.37
#> 126  1.32
#> 127  1.27
#> 128  1.23
#> 129  1.18
#> 130  1.14
#> 131  1.1
#> 132  1.06
#> 133  1.02
#> 134  0.989
#> 135  0.954
#> 136  0.921
#> 137  0.889
#> 138  0.858
#> 139  0.828
```

```
#> 140  0.799
#> 141  0.772
#> 142  0.745
#> 143  0.719
#> 144  0.695
#> 145  0.671
#> 146  0.648
#> 147  0.626
#> 148  0.605
#> 149  0.584
#> 150  0.564
#> 151  0.545
#> 152  0.527
#> 153  0.509
#> 154  0.492
#> 155  0.476
#> 156  0.46
#> 157  0.444
#> 158  0.43
#> 159  0.415
#> 160  0.402
#> 161  0.388
#> 162  0.375
#> 163  0.363
#> 164  0.351
#> 165  0.339
#> 166  0.328
#> 167  0.318
#> 168  0.307
#> 169  0.297
#> 170  0.287
#> 171  0.278
#> 172  0.269
#> 173  0.26
#> 174  0.252
#> 175  0.244
#> 176  0.236
#> 177  0.228
#> 178  0.221
#> 179  0.214
#> 180  0.207
#> 181  0.2
#> 182  0.194
#> 183  0.187
#> 184  0.181
```

```
#> 185  0.176
#> 186  0.17
#> 187  0.165
#> 188  0.159
#> 189  0.154
#> 190  0.149
#> 191  0.145
#> 192  0.14
#> 193  0.136
#> 194  0.131
#> 195  0.127
#> 196  0.123
#> 197  0.119
#> 198  0.115
#> 199  0.112
#> 200  0.108
#> 201  0.105
#> 202  0.102
#> 203  0.0983
#> 204  0.0952
#> 205  0.0923
#> 206  0.0894
#> 207  0.0866
#> 208  0.0838
#> 209  0.0812
#> 210  0.0787
#> 211  0.0762
#> 212  0.0739
#> 213  0.0716
#> 214  0.0693
#> 215  0.0672
#> 216  0.0651
#> 217  0.0631
#> 218  0.0611
#> 219  0.0592
#> 220  0.0574
#> 221  0.0556
#> 222  0.0539
#> 223  0.0522
#> 224  0.0506
#> 225  0.0491
#> 226  0.0476
#> 227  0.0461
#> 228  0.0447
#> 229  0.0433
```

```
#> 230  0.042
#> 231  0.0407
#> 232  0.0394
#> 233  0.0382
#> 234  0.0371
#> 235  0.0359
#> 236  0.0348
#> 237  0.0338
#> 238  0.0327
#> 239  0.0317
#> 240  0.0308
#> 241  0.0298
#> 242  0.0289
#> 243  0.028
#> 244  0.0272
#> 245  0.0263
#> 246  0.0255
#> 247  0.0248
#> 248  0.024
#> 249  0.0233
#> 250  0.0226
#> 251  0.0219
#> 252  0.0212
#> 253  0.0206
#> 254  0.02
#> 255  0.0194
#> 256  0.0188
#> 257  0.0182
#> 258  0.0177
#> 259  0.0171
#> 260  0.0166
#> 261  0.0161
#> 262  0.0156
#> 263  0.0151
#> 264  0.0147
#> 265  0.0142
#> 266  0.0138
#> 267  0.0134
#> 268  0.013
#> 269  0.0126
#> 270  0.0122
#> 271  0.0119
#> 272  0.0115
#> 273  0.0112
#> 274  0.0108
```

```
#> 275  0.0105
#> 276  0.0102
#> 277  0.00988
#> 278  0.00959
#> 279  0.0093
#> 280  0.00902
#> 281  0.00875
#> 282  0.00849
#> 283  0.00824
#> 284  0.00799
#> 285  0.00775
#> 286  0.00752
#> 287  0.0073
#> 288  0.00708
#> 289  0.00687
#> 290  0.00666
#> 291  0.00647
#> 292  0.00627
#> 293  0.00609
#> 294  0.00591
#> 295  0.00573
#> 296  0.00556
#> 297  0.0054
#> 298  0.00524
#> 299  0.00508
#> 300  0.00493
#> 301  0.00478
#> 302  0.00464
#> 303  0.0045
#> 304  0.00437
#> 305  0.00424
#> 306  0.00412
#> 307  0.00399
#> 308  0.00388
#> 309  0.00376
#> 310  0.00365
#> 311  0.00354
#> 312  0.00344
#> 313  0.00334
#> 314  0.00324
#> 315  0.00314
#> 316  0.00305
#> 317  0.00296
#> 318  0.00287
#> 319  0.00279
```

```
#> 320  0.00271
#> 321  0.00263
#> 322  0.00255
#> 323  0.00248
#> 324  0.0024
#> 325  0.00233
#> 326  0.00226
#> 327  0.0022
#> 328  0.00213
#> 329  0.00207
#> 330  0.00201
#> 331  0.00195
#> 332  0.00189
#> 333  0.00184
#> 334  0.00178
#> 335  0.00173
#> 336  0.00168
#> 337  0.00163
#> 338  0.00158
#> 339  0.00154
#> 340  0.00149
#> 341  0.00145
#> 342  0.00141
#> 343  0.00137
#> 344  0.00133
#> 345  0.00129
#> 346  0.00125
#> 347  0.00121
#> 348  0.00118
#> 349  0.00114
#> 350  0.00111
#> 351  0.00108
#> 352  0.00105
#> 353  0.00102
#> 354  0.000987
#> 355  0.000958
#> 356  0.000931
#> 357  0.000904
#> 358  0.000877
#> 359  0.000852
#> 360  0.000827
#> 361  0.000803
#> 362  0.00078
#> 363  0.000757
#> 364  0.000735
```

```
#> 365  0.000714
#> 366  0.000693
#> 367  0.000673
#> 368  0.000654
#> 369  0.000635
#> 370  0.000617
#> 371  0.000599
#> 372  0.000581
#> 373  0.000565
#> 374  0.000548
#> 375  0.000532
#> 376  0.000517
#> 377  0.000502
#> 378  0.000488
#> 379  0.000474
#> 380  0.00046
#> 381  0.000447
#> 382  0.000434
#> 383  0.000421
#> 384  0.000409
#> 385  0.000397
#> 386  0.000386
#> 387  0.000375
#> 388  0.000364
#> 389  0.000354
#> 390  0.000343
#> 391  0.000334
#> 392  0.000324
#> 393  0.000315
#> 394  0.000306
#> 395  0.000297
#> 396  0.000288
#> 397  0.00028
#> 398  0.000272
#> 399  0.000264
#> 400  0.000257
#> 401  0.000249
#> 402  0.000242
#> 403  0.000235
#> 404  0.000228
#> 405  0.000222
#> 406  0.000216
#> 407  0.000209
#> 408  0.000203
#> 409  0.000198
```

```
#> 410  0.000192
#> 411  0.000186
#> 412  0.000181
#> 413  0.000176
#> 414  0.000171
#> 415  0.000166
#> 416  0.000161
#> 417  0.000157
#> 418  0.000152
#> 419  0.000148
#> 420  0.000144
#> 421  0.00014
#> 422  0.000136
#> 423  0.000132
#> 424  0.000128
#> 425  0.000124
#> 426  0.000121
#> 427  0.000117
#> 428  0.000114
#> 429  0.000111
#> 430  0.000108
#> 431  0.000105
#> 432  0.000102
#> 433  9.87e-05
#> 434  9.59e-05
#> 435  9.32e-05
#> 436  9.06e-05
#> 437  8.8e-05
#> 438  8.55e-05
#> 439  8.31e-05
#> 440  8.07e-05
#> 441  7.84e-05
#> 442  7.62e-05
#> 443  7.41e-05
#> 444  7.2e-05
#> 445  6.99e-05
#> 446  6.79e-05
#> 447  6.6e-05
#> 448  6.41e-05
#> 449  6.23e-05
#> 450  6.06e-05
#> 451  5.89e-05
#> 452  5.72e-05
#> 453  5.56e-05
#> 454  5.4e-05
```

```
#> 455  5.25e-05
#> 456  5.1e-05
#> 457  4.96e-05
#> 458  4.82e-05
#> 459  4.68e-05
#> 460  4.55e-05
#> 461  4.42e-05
#> 462  4.3e-05
#> 463  4.18e-05
#> 464  4.06e-05
#> 465  3.94e-05
#> 466  3.83e-05
#> 467  3.72e-05
#> 468  3.62e-05
#> 469  3.52e-05
#> 470  3.42e-05
#> 471  3.32e-05
#> 472  3.23e-05
#> 473  3.14e-05
#> 474  3.05e-05
#> 475  2.96e-05
#> 476  2.88e-05
#> 477  2.8e-05
#> 478  2.72e-05
#> 479  2.65e-05
#> 480  2.57e-05
#> 481  2.5e-05
#> 482  2.43e-05
#> 483  2.36e-05
#> 484  2.29e-05
#> 485  2.23e-05
#> 486  2.17e-05
#> 487  2.11e-05
#> 488  2.05e-05
#> 489  1.99e-05
#> 490  1.94e-05
#> 491  1.88e-05
#> 492  1.83e-05
#> 493  1.78e-05
#> 494  1.73e-05
#> 495  1.68e-05
#> 496  1.63e-05
#> 497  1.59e-05
#> 498  1.54e-05
#> 499  1.5e-05
```

```
#> 500  1.46e-05
```

These two expression are equivalent, with the first being the long version natural way of doing it in **PyTorch**. The second is using the generics in R for subtraction, multiplication and scalar conversion.

```
param$data <- torch$sub(param$data,
                        torch$mul(param$grad$float(),
                          torch$scalar_tensor(learning_rate)))
}

param$data <- param$data - param$grad * learning_rate
```

Chapter 9

Neural Networks 2

Source: https://github.com/jcjohnson/pytorch-examples#pytorch-nn

9.1 nn2 1

9.2 nn2 2

Part VI

Image Recognition

Part VII

PyTorch and R data structures

Chapter 10

Working with data.frame

10.1 Load PyTorch libraries

```
library(rTorch)

torch       <- import("torch")
torchvision <- import("torchvision")
nn          <- import("torch.nn")
transforms  <- import("torchvision.transforms")
dsets       <- import("torchvision.datasets")
builtins    <- import_builtins()
np          <- import("numpy")
```

10.2 Dataset iteration batch settings

```
# folders where the images are located
train_data_path = '~/mnist_png_full/training/'
test_data_path  = '~/mnist_png_full/testing/'
```

```
# read the datasets without normalization
train_dataset = torchvision$datasets$ImageFolder(root = train_data_path,
    transform = torchvision$transforms$ToTensor()
)

print(train_dataset)
#> Dataset ImageFolder
#>     Number of datapoints: 60000
```

```
#>     Root location: /home/msfz751/mnist_png_full/training/
```

10.3 Summary statistics for tensors

10.4 using `data.frame`

```
library(tictoc)
tic()

fun_list <- list(
    size  = c("size"),
    numel = c("numel"),
    sum   = c("sum",    "item"),
    mean  = c("mean",   "item"),
    std   = c("std",    "item"),
    med   = c("median", "item"),
    max   = c("max",    "item"),
    min   = c("min",    "item")
    )

idx <- seq(0L, 599L)    # how many samples

fun_get_tensor <- function(x) py_get_item(train_dataset, x)[[1]]

stat_fun <- function(x, str_fun) {
  fun_var <- paste0("fun_get_tensor(x)", "$", str_fun, "()")
  sapply(idx, function(x)
    ifelse(is.numeric(eval(parse(text = fun_var))),  # size return chracater
           eval(parse(text = fun_var)),              # all else are numeric
           as.character(eval(parse(text = fun_var)))))
}

df <- data.frame(ridx = idx+1,      # index number for the sample
  do.call(data.frame,
          lapply(
              sapply(fun_list, function(x) paste(x, collapse = "()$")),
              function(y) stat_fun(1, y)
          )
  )
)
head(df)
#>   ridx                    size numel sum  mean   std med max min
```

```
#> 1    1 torch.Size([3, 28, 28])  2352 366 0.156 0.329   0   1   0
#> 2    2 torch.Size([3, 28, 28])  2352 284 0.121 0.297   0   1   0
#> 3    3 torch.Size([3, 28, 28])  2352 645 0.274 0.420   0   1   0
#> 4    4 torch.Size([3, 28, 28])  2352 410 0.174 0.355   0   1   0
#> 5    5 torch.Size([3, 28, 28])  2352 321 0.137 0.312   0   1   0
#> 6    6 torch.Size([3, 28, 28])  2352 654 0.278 0.421   0   1   0
toc()
#> 12.504 sec elapsed
# 59    1.663s
#   599  13.5s
#  5999  54.321 sec; 137.6s
# 59999 553.489 sec elapsed
```

Chapter 11

Working with data.table

11.1 Load PyTorch libraries

```
library(rTorch)

torch       <- import("torch")
torchvision <- import("torchvision")
nn          <- import("torch.nn")
transforms  <- import("torchvision.transforms")
dsets       <- import("torchvision.datasets")
builtins    <- import_builtins()
np          <- import("numpy")

## Dataset iteration batch settings
# folders where the images are located
train_data_path = '~/mnist_png_full/training/'
test_data_path  = '~/mnist_png_full/testing/'

# read the datasets without normalization
train_dataset = torchvision$datasets$ImageFolder(root = train_data_path,
    transform = torchvision$transforms$ToTensor()
)

print(train_dataset)
#> Dataset ImageFolder
#>     Number of datapoints: 60000
#>     Root location: /home/msfz751/mnist_png_full/training/
```

11.1.1 Using 'data.table

```
library(data.table)
library(tictoc)
tic()

fun_list <- list(
    numel = c("numel"),
    sum   = c("sum",    "item"),
    mean  = c("mean",   "item"),
    std   = c("std",    "item"),
    med   = c("median", "item"),
    max   = c("max",    "item"),
    min   = c("min",    "item")
    )

idx <- seq(0L, 5999L)

fun_get_tensor <- function(x) py_get_item(train_dataset, x)[[1]]

stat_fun <- function(x, str_fun) {
  fun_var <- paste0("fun_get_tensor(x)", "$", str_fun, "()")
  sapply(idx, function(x)
    ifelse(is.numeric(eval(parse(text = fun_var))),  # size return chracater
           eval(parse(text = fun_var)),              # all else are numeric
           as.character(eval(parse(text = fun_var)))))
}

dt <- data.table(ridx = idx+1,
  do.call(data.table,
          lapply(
            sapply(fun_list, function(x) paste(x, collapse = "()$")),
            function(y) stat_fun(1, y)
          )
  )
)

head(dt)
#>    ridx numel sum  mean   std med max min
#> 1:    1  2352 366 0.156 0.329   0   1   0
#> 2:    2  2352 284 0.121 0.297   0   1   0
#> 3:    3  2352 645 0.274 0.420   0   1   0
#> 4:    4  2352 410 0.174 0.355   0   1   0
#> 5:    5  2352 321 0.137 0.312   0   1   0
#> 6:    6  2352 654 0.278 0.421   0   1   0
```

```
toc()
#> 112.584 sec elapsed

#   60   1.266 sec elapsed
#  600   11.798 sec elapsed; 12.9s
# 6000  119.256 sec elapsed; 132.9s
# 1117.619 sec elapsed
```

Appendix A

Statistical Background

A.1 Basic statistical terms

A.1.1 Mean

The mean is the most commonly reported measure of center. It is commonly called the "average" though this term can be a little ambiguous. The mean is the sum of all of the data elements divided by how many elements there are. If we have n data points, the mean is given by:

$$Mean = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

A.1.2 Median

The median is calculated by first sorting a variable's data from smallest to largest. After sorting the data, the middle element in the list is the **median**. If the middle falls between two values, then the median is the mean of those two values.

A.1.3 Standard deviation

We will next discuss the **standard deviation** of a sample dataset pertaining to one variable. The formula can be a little intimidating at first but it is important to remember that it is essentially a measure of how far to expect a given data value is from its mean:

$$Standard\,deviation = \sqrt{\frac{(x_1 - Mean)^2 + (x_2 - Mean)^2 + \cdots + (x_n - Mean)^2}{n-1}}$$

A.1.4 Five-number summary

The **five-number summary** consists of five values: minimum, first quantile AKA 25^{th} percentile, second quantile AKA median AKA 50^{th} percentile, third quantile AKA 75^{th}, and maximum. The quantiles are calculated as

- first quantile (Q_1): the median of the first half of the sorted data
- third quantile (Q_3): the median of the second half of the sorted data

The *interquartile range* is defined as $Q_3 - Q_1$ and is a measure of how spread out the middle 50% of values is. The five-number summary is not influenced by the presence of outliers in the ways that the mean and standard deviation are. It is, thus, recommended for skewed datasets.

A.1.5 Distribution

The **distribution** of a variable/dataset corresponds to generalizing patterns in the dataset. It often shows how frequently elements in the dataset appear. It shows how the data varies and gives some information about where a typical element in the data might fall. Distributions are most easily seen through data visualization.

A.1.6 Outliers

Outliers correspond to values in the dataset that fall far outside the range of "ordinary" values. In regards to a boxplot (by default), they correspond to values below $Q_1 - (1.5 * IQR)$ or above $Q_3 + (1.5 * IQR)$.

Note that these terms (aside from **Distribution**) only apply to quantitative variables.

Bibliography

Cordes, D. and Brown, M. (1991). The literate-programming paradigm. *Computer*, 24(6):52–61.

Knuth, D. E. (1983). Literate programming. *The Computer Journal*, 27(Issue 2):97–111.