A Minimal rTorch Tutorial

Alfonso R. Reyes

2019-08-10

Contents

Chapter 1

Prerequisites

You need two things to get `rTorch` working:

1. Python Anaconda installed.
2. Install `rTorch` from CRAN or GitHub.

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

```
install.packages("rTorch")
# or the development version
# devtools::install_github("f0nzie/rTorch")
```

> Note. It is not mandatory to have a previously Python environment created with Anaconda where PyTorch and TorchVision have been installed. You could also installed directly from the R console in similar fashion as TensorFlow using the function `install_pytorch`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 2

Introduction

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 2. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter @ref(linear_algebra).

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 2.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 2.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2019) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

2.1 Installation

`rTorch` is available in GitHub only at this moment.

Install `rTorch` with:

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

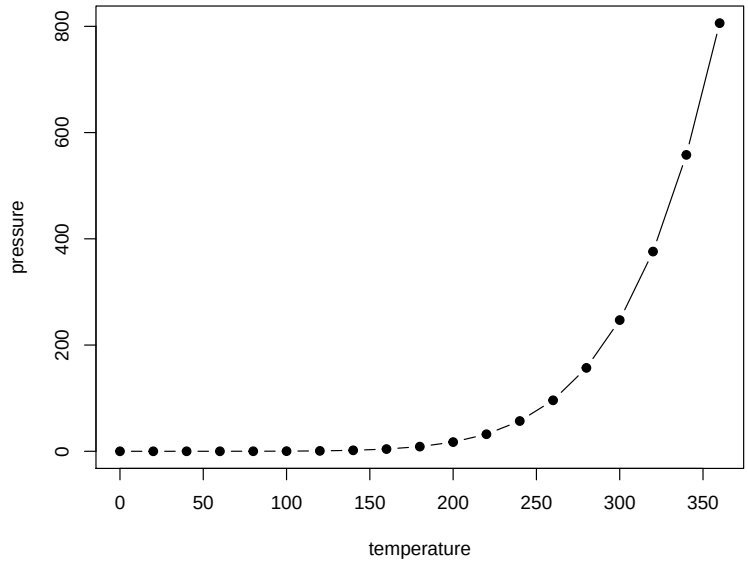


Figure 2.1: Here is a nice figure!

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

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

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

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

The automatic installation, like in `rtensorflow`, may be available later.

Note. `matplotlib` and `pandas` are not really necessary, but I was asked if `matplotlib` or `pandas` would in PyTorch, that I decided to put them for testing and experimentation. They both work.

2.2 Matrices and Linear Algebra

There are five major type of Tensors in PyTorch

```
library(rTorch)

bt <- torch$ByteTensor(3L, 3L)
ft <- torch$FloatTensor(3L, 3L)
```

Table 2.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

```
dt <- torch$DoubleTensor(3L, 3L)
lt <- torch$LongTensor(3L, 3L)
Bt <- torch$BoolTensor(5L, 5L)

ft
```

```
## tensor([[-6.3276e-21,  4.5744e-41, -6.3276e-21],
##         [ 4.5744e-41,  4.1075e+18,  9.2737e-41],
##         [ 2.1163e-37,  2.1214e-37,  1.9304e-37]])
```

```
dt
```

```
## tensor([[6.9272e-310, 4.6816e-310, 4.6816e-310],
##         [6.9272e-310, 8.3991e-323,  0.0000e+00],
##         [4.6816e-310, 4.6816e-310, 4.6816e-310]], dtype=torch.float64)
```

```
Bt
```

```
## tensor([[ True,  True,  True,  True,  True],
##         [ True, False, False,  True,  True],
##         [ True,  True,  True,  True, False],
```

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

A 4D tensor like in MNIST hand-written digits recognition dataset:

```
mnist_4d <- torch$FloatTensor(60000L, 3L, 28L, 28L)

# size
mnist_4d$size()
```

```
## torch.Size([60000, 3, 28, 28])
```

```
# length
length(mnist_4d)
```

```
## [1] 141120000
```

```
# shape, like in numpy
mnist_4d$shape
```

```
## torch.Size([60000, 3, 28, 28])
```

```
# number of elements
mnist_4d$numel()
```

```
## [1] 141120000
```

A 3D tensor:

```
ft3d <- torch$FloatTensor(4L, 3L, 2L)
ft3d
```

```
## tensor([[[2.5223e-44, 0.0000e+00],
##          [0.0000e+00, 0.0000e+00],
##          [2.3557e+30, 3.0915e-41]],
##
##         [[2.3557e+30, 3.0915e-41],
##          [2.3557e+30, 3.0915e-41],
##          [1.1412e+30, 3.0915e-41]],
##
##         [[1.1412e+30, 3.0915e-41],
##          [1.1412e+30, 3.0915e-41],
##          [1.1412e+30, 3.0915e-41]],
##
##         [[1.1412e+30, 3.0915e-41],
##          [1.1412e+30, 3.0915e-41],
##          [1.1412e+30, 3.0915e-41]]])
```

```
# get first element in a tensor
ft3d[1, 1, 1]
```

```
## tensor(2.5223e-44)
```

```
bt
```

```
## tensor([[160,  12, 239],
##         [157, 132, 127],
##         [  0,   0, 160]], dtype=torch.uint8)
```

```
# [torch.ByteTensor of size 3x3]
```

```
ft
```

```
## tensor([[-6.3276e-21,  4.5744e-41, -6.3276e-21],
##         [ 4.5744e-41,  4.1075e+18,  9.2737e-41],
##         [ 2.1163e-37,  2.1214e-37,  1.9304e-37]])
```

```
# [torch.FloatTensor of size 3x3]
```

```
# create a tensor with a value
torch$full(list(2L, 3L), 3.141592)
```

```
## tensor([[3.1416, 3.1416, 3.1416],
##         [3.1416, 3.1416, 3.1416]])
```

Chapter 3

Lessons Learned

Here is a review of existing methods.

```
library(rTorch)
```

3.1 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
```

3.2 How to iterate a generator

3.2.1 Using `enumerate` and `iterate`

```
py = import_builtins()

xx = py$enumerate(x_train)
xit = iterate(xx, 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])
```

3.2.2 Using a `for-loop`

3.3 Zero gradient

3.4 Transform a tensor

3.5 Build a model class

3.6 Convert a tensor to numpy object

3.7 Convert a numpy object to an R object

Chapter 4

Tensors

We describe our methods in this chapter.

4.1 R code

4.1.1 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.

4.1.2 Datasets

```
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$randn(D_in, H, device=device)
w2 = torch$randn(H, D_out, device=device)
```

4.1.3 Run the model

```
learning_rate = 1e-6
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    17973400
## 6    15345387
## 7    11953077
## 8    8557820
## 9    5777508
## 10   3791835
## 11   2494379
## 12   1679618
## 13   1176170
## 14   858874.1
## 15   654740.2
## 16   517358.8
## 17   421627.9
## 18   351478.8
## 19   298321
## 20   256309.5
## 21   222513
## 22   194530.1
## 23   171047.8
## 24   151091.8
## 25   134001
## 26   119256.3
## 27   106430.9
## 28   95219.91
## 29   85393.08
## 30   76738.77
## 31   69098.89
## 32   62340.1
## 33   56344.36
## 34   51009.01
## 35   46249.34
## 36   41991.58
## 37   38181.8
## 38   34770.48
## 39   31705.34
## 40   28945.98
## 41   26458.03
## 42   24211.19
## 43   22179.04
## 44   20333.61
## 45   18658.66
## 46   17137.63
## 47   15753.23
## 48   14493.79
## 49   13346.74
```

```
## 50   12301.08
```

Chapter 5

Linear Algebra with Torch

```
library(rTorch)
```

5.1 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.)
```

5.2 Vectors

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

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

```
# row-vector
(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
```

```
torch$as_tensor(mr)
```

```
## tensor([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10]], dtype=torch.int32)
```

```
torch$as_tensor(mr)$shape
```

```
## torch.Size([1, 10])
```

```
# column-vector
(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
```

```
torch$as_tensor(mc)
```

```
## tensor([[ 1],
##         [ 2],
##         [ 3],
##         [ 4],
##         [ 5],
##         [ 6],
##         [ 7],
##         [ 8],
##         [ 9],
##         [10]], dtype=torch.int32)
```

```
torch$as_tensor(mc)$shape
```

```
## torch.Size([10, 1])
```

5.3 Matrices

```
(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
```

```
(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)
```

```
torch$as_tensor(m1)$shape
```

```
## torch.Size([3, 8])
```

```
torch$as_tensor(m1)$size()
```

```
## torch.Size([3, 8])
```

```
dim(torch$as_tensor(m1))
```

```
## [1] 3 8
```

```
length(torch$as_tensor(m1))
```

```
## [1] 24
```

```
(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
```

```
(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)
```

```
t2$shape
```

```
## torch.Size([10, 10])
```

```
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)
```

5.4 3D+ tensors

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

```
## tensor([[[0.5548, 0.9973, 0.6722,  ..., 0.4775, 0.6007, 0.4875],
##          [0.1537, 0.6212, 0.6013,  ..., 0.2184, 0.3347, 0.5631],
##          [0.0460, 0.1544, 0.4286,  ..., 0.9303, 0.4223, 0.5124],
##          ...,
##          [0.9737, 0.0102, 0.8282,  ..., 0.5079, 0.3078, 0.0924],
##          [0.3454, 0.9608, 0.2808,  ..., 0.2469, 0.3147, 0.4546],
##          [0.3443, 0.8095, 0.7795,  ..., 0.5723, 0.8290, 0.3287]],
##
##         [[0.6399, 0.2090, 0.0342,  ..., 0.3827, 0.3678, 0.9032],
##          [0.5105, 0.7987, 0.6801,  ..., 0.3994, 0.0647, 0.3114],
##          [0.3356, 0.4653, 0.1321,  ..., 0.3684, 0.5010, 0.4759],
```

```
##           ...,
##           [0.0171, 0.2603, 0.7928,  ..., 0.6749, 0.5378, 0.5613],
##           [0.7369, 0.1967, 0.8680,  ..., 0.1407, 0.8730, 0.0515],
##           [0.2829, 0.9587, 0.6520,  ..., 0.0570, 0.3087, 0.0332]],
##
##          [[0.4107, 0.7514, 0.7483,  ..., 0.5357, 0.1396, 0.1590],
##           [0.0334, 0.1690, 0.8897,  ..., 0.2856, 0.0397, 0.6905],
##           [0.9382, 0.2469, 0.5983,  ..., 0.2219, 0.5791, 0.4991],
##           ...,
##           [0.4591, 0.3552, 0.2316,  ..., 0.0053, 0.4699, 0.0691],
##           [0.0557, 0.5610, 0.8311,  ..., 0.9081, 0.7571, 0.6281],
##           [0.3794, 0.2286, 0.5723,  ..., 0.6046, 0.7877, 0.6717]]])
```

```
img$shape
```

```
## torch.Size([3, 28, 28])
```

```
img[1, 1, 1]
```

```
## tensor(0.5548)
```

```
img[3, 28, 28]
```

```
## tensor(0.6717)
```

5.5 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
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
```

```
(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)
```

```
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
```

5.6 Vectors, special case of a matrix

```
m2 <- matrix(0:99, ncol = 10)
(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
(v1 <- m2[, 1])
```

```
##  [1] 0 1 2 3 4 5 6 7 8 9
```

```
(v2 <- m2[10, ])
```

```
##  [1]  9 19 29 39 49 59 69 79 89 99
```

```
# PyTorch

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)
```

5.7 Tensor addition

```
(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.]])
```

```
(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.]])
```

```
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)
```

5.8 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.],
##         [1., 1., 1., 1.]])
```

5.9 Multiplying tensors

$$A * B = B * A$$

```
(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.]])
```

```
(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.]])
```

```
(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)
```

5.10 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)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): RuntimeError: dot: Expect
```

```
# 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: Expected 1-D arg
```

```
o1$dot(o2)
```

```
## Error in py_call_impl(callable, dots$args, dots$keywords): RuntimeError: dot: Expect
```

```
# 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([[ 0.6558,  0.7177,  0.0362, -0.1869],
##         [-0.1548, -0.7869,  0.0805,  0.3147]])
```

```
a$matmul(b)
```

```
## tensor([[ 0.6558,  0.7177,  0.0362, -0.1869],
##         [-0.1548, -0.7869,  0.0805,  0.3147]])
```

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

```
abt <- torch$mm(a, b)$transpose(dim0=0L, dim1=1L)
abt
```

```
## tensor([[ 0.6558, -0.1548],
##         [ 0.7177, -0.7869],
##         [ 0.0362,  0.0805],
##         [-0.1869,  0.3147]])
```

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

btat <- torch$matmul(bt, at)
btat
```

```
## tensor([[ 0.6558, -0.1548],
##         [ 0.7177, -0.7869],
##         [ 0.0362,  0.0805],
##         [-0.1869,  0.3147]])
```

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

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

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

Chapter 6

Linear Regression

6.1 Case 1: simple linear regression

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

6.2 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)
)
```

```
library(rTorch)

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

torch$manual_seed(123)
```

```
## <torch._C.Generator>
```

```
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)
## )
```

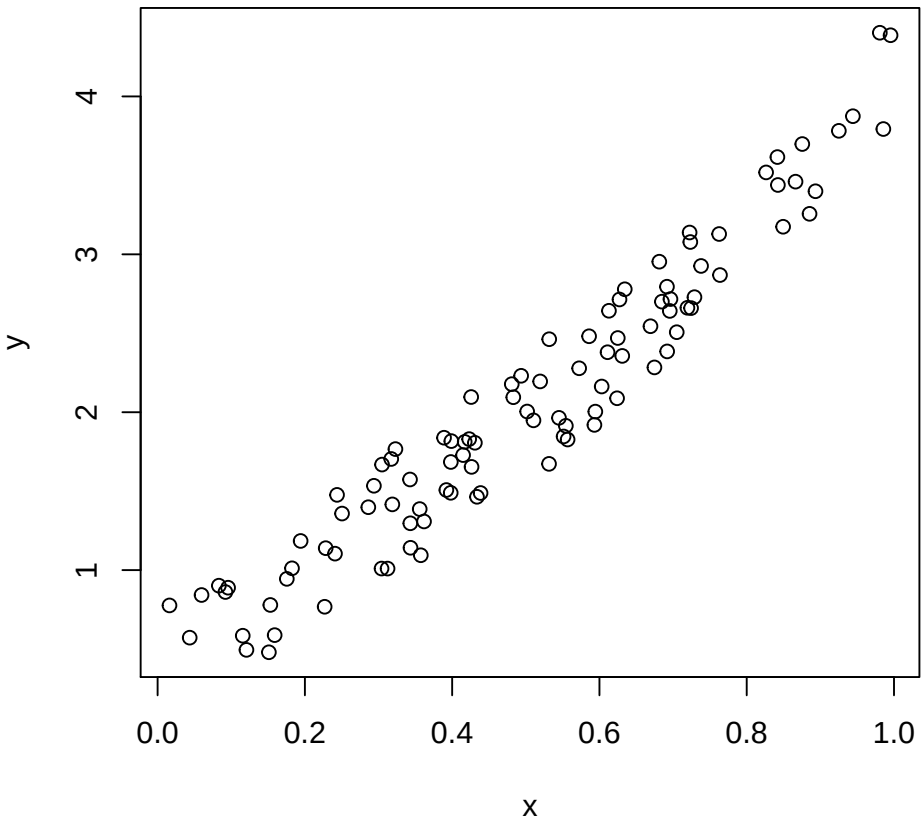
6.3 Datasets

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)
```



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

6.3.1 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]])
```

6.4 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()
```

6.5 Training

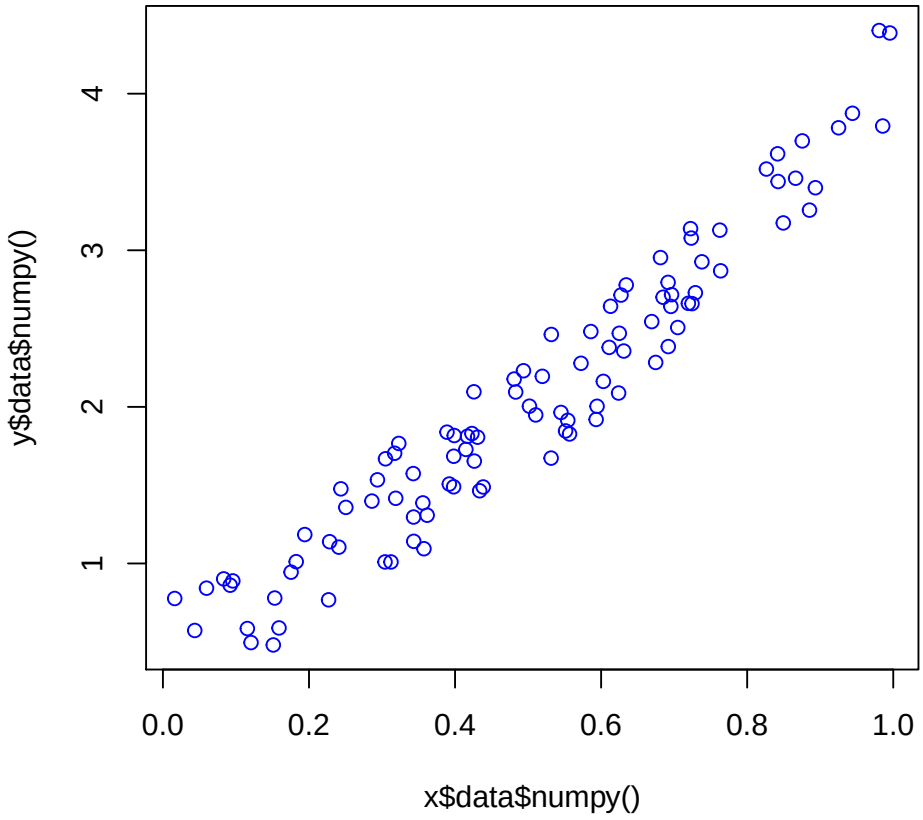
Now let's start our training process. With an epoch of 250, you will iterate our data to find the best value for our hyperparameters.

```
# 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")
   }
}
```

6.6 Result

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.

6.7 Case 2: Rainfall

```
library(rTorch)
```

6.8 Select device

```
torch$manual_seed(0)
```

```
## <torch._C.Generator>
```

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

6.9 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')
```

6.10 Convert 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)
```

6.11 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))
}
```

6.12 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.

6.13 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()))
}
```

```
# 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.

6.14 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.

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.])
```

6.15 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>)
```

6.16 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
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)
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.]])
```

Chapter 7

Logistic Regression

```
library(rTorch)

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

torch$set_default_dtype(torch$float)
```

7.0.1 Hyperparameters

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

7.0.2 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
```

7.0.3 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)
```

7.0.4 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: 2.193622
## Epoch [2/5], Step [200/600], Loss: 2.095055
## Epoch [2/5], Step [300/600], Loss: 1.993158
## Epoch [2/5], Step [400/600], Loss: 1.914482
## Epoch [2/5], Step [500/600], Loss: 1.828060
## Epoch [2/5], Step [600/600], Loss: 1.821115
## Epoch [3/5], Step [100/600], Loss: 1.732071
## Epoch [3/5], Step [200/600], Loss: 1.653721
## Epoch [3/5], Step [300/600], Loss: 1.592439
## Epoch [3/5], Step [400/600], Loss: 1.525097
## Epoch [3/5], Step [500/600], Loss: 1.436987
## Epoch [3/5], Step [600/600], Loss: 1.502717
## Epoch [4/5], Step [100/600], Loss: 1.446447
## Epoch [4/5], Step [200/600], Loss: 1.366274
## Epoch [4/5], Step [300/600], Loss: 1.341085
## Epoch [4/5], Step [400/600], Loss: 1.280446
## Epoch [4/5], Step [500/600], Loss: 1.181919
## Epoch [4/5], Step [600/600], Loss: 1.288031
## Epoch [5/5], Step [100/600], Loss: 1.259706
## Epoch [5/5], Step [200/600], Loss: 1.173801
```

```
## Epoch [5/5], Step [300/600], Loss: 1.176926
## Epoch [5/5], Step [400/600], Loss: 1.122017
## Epoch [5/5], Step [500/600], Loss: 1.010820
## Epoch [5/5], Step [600/600], Loss: 1.138838
## Epoch [6/5], Step [100/600], Loss: 1.130953
## Epoch [6/5], Step [200/600], Loss: 1.039641
## Epoch [6/5], Step [300/600], Loss: 1.063141
## Epoch [6/5], Step [400/600], Loss: 1.013657
## Epoch [6/5], Step [500/600], Loss: 0.890745
## Epoch [6/5], Step [600/600], Loss: 1.030636
```

7.0.5 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: 83.730000 %
```

7.0.6 Save the model

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

Chapter 8

Neural Networks

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

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

8.1 In R

8.1.1 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.
```

8.1.2 Create datasets

```
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)
```

8.1.3 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),
          torch$nn$ReLU(),
          torch$nn$Linear(H, D_out))$to(device)
```

8.1.4 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')
```

8.1.5 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    628.2839
## 2    584.9592
## 3    546.728
## 4    512.6885
## 5    482.1306
## 6    454.6265
## 7    429.5551
## 8    406.399
## 9    384.6441
## 10   364.3125
## 11   345.3866
## 12   327.6258
## 13   310.8002
## 14   294.8379
## 15   279.7325
## 16   265.308
```

```
## 17   251.6231
## 18   238.5361
## 19   226.0232
## 20   214.1025
## 21   202.7003
## 22   191.8059
## 23   181.4473
## 24   171.5948
## 25   162.2114
## 26   153.2922
## 27   144.8526
## 28   136.86
## 29   129.2786
## 30   122.0849
## 31   115.2736
## 32   108.8145
## 33   102.702
## 34   96.9281
## 35   91.47466
## 36   86.3278
## 37   81.47502
## 38   76.89146
## 39   72.56895
## 40   68.47774
## 41   64.62608
## 42   60.99714
## 43   57.56915
## 44   54.3427
## 45   51.30913
## 46   48.4516
## 47   45.7645
## 48   43.2405
## 49   40.86606
## 50   38.63006
## 51   36.52527
## 52   34.54914
## 53   32.69345
## 54   30.9498
## 55   29.30734
## 56   27.76275
## 57   26.31089
## 58   24.9431
## 59   23.6538
## 60   22.43881
## 61   21.29258
## 62   20.21278
```

```
## 63   19.19288
## 64   18.23302
## 65   17.32939
## 66   16.47696
## 67   15.67115
## 68   14.90951
## 69   14.19054
## 70   13.51087
## 71   12.86819
## 72   12.26046
## 73   11.68514
## 74   11.14105
## 75   10.62579
## 76   10.13727
## 77   9.67443
## 78   9.235745
## 79   8.81936
## 80   8.423415
## 81   8.047976
## 82   7.691305
## 83   7.352424
## 84   7.030395
## 85   6.724676
## 86   6.434433
## 87   6.158278
## 88   5.895441
## 89   5.645772
## 90   5.408593
## 91   5.182788
## 92   4.967783
## 93   4.762821
## 94   4.567942
## 95   4.381676
## 96   4.203955
## 97   4.034254
## 98   3.872373
## 99   3.717669
## 100  3.569852
## 101  3.42879
## 102  3.293934
## 103  3.165006
## 104  3.041733
## 105  2.923832
## 106  2.810992
## 107  2.702966
## 108  2.599546
```

```
## 109  2.500533
## 110  2.405609
## 111  2.314677
## 112  2.227567
## 113  2.143984
## 114  2.063824
## 115  1.986898
## 116  1.913161
## 117  1.842398
## 118  1.77456
## 119  1.709334
## 120  1.64675
## 121  1.586662
## 122  1.528932
## 123  1.473498
## 124  1.420194
## 125  1.369045
## 126  1.319939
## 127  1.272713
## 128  1.227352
## 129  1.183716
## 130  1.141713
## 131  1.101289
## 132  1.062379
## 133  1.024975
## 134  0.9889802
## 135  0.9543175
## 136  0.9209599
## 137  0.8888108
## 138  0.8578573
## 139  0.828075
## 140  0.7993811
## 141  0.7717422
## 142  0.7451235
## 143  0.7194628
## 144  0.6947174
## 145  0.6708718
## 146  0.6479479
## 147  0.6259133
## 148  0.6046655
## 149  0.5841969
## 150  0.5644467
## 151  0.5453879
## 152  0.5270226
## 153  0.509312
## 154  0.4922202
```

```
## 155  0.4757309
## 156  0.4598286
## 157  0.444492
## 158  0.4296912
## 159  0.4153878
## 160  0.4015924
## 161  0.3882794
## 162  0.3754345
## 163  0.3630294
## 164  0.3510535
## 165  0.3394946
## 166  0.3283299
## 167  0.3175527
## 168  0.3071378
## 169  0.2970789
## 170  0.2873671
## 171  0.2779845
## 172  0.2689149
## 173  0.2601575
## 174  0.2516952
## 175  0.2435159
## 176  0.2356067
## 177  0.2279707
## 178  0.2206163
## 179  0.2135031
## 180  0.2066278
## 181  0.199988
## 182  0.1935741
## 183  0.1873666
## 184  0.1813661
## 185  0.1755629
## 186  0.1699517
## 187  0.1645233
## 188  0.1592781
## 189  0.1542067
## 190  0.149301
## 191  0.1445545
## 192  0.1399648
## 193  0.1355246
## 194  0.1312279
## 195  0.1270718
## 196  0.1230509
## 197  0.1191625
## 198  0.115399
## 199  0.1117588
## 200  0.1082348
```

```
## 201  0.1048254
## 202  0.1015256
## 203  0.09833349
## 204  0.09524446
## 205  0.0922533
## 206  0.08935806
## 207  0.08655693
## 208  0.08384982
## 209  0.081232
## 210  0.07869583
## 211  0.07624327
## 212  0.0738695
## 213  0.07157086
## 214  0.06934651
## 215  0.0671912
## 216  0.06510548
## 217  0.0630878
## 218  0.0611335
## 219  0.05924014
## 220  0.05740607
## 221  0.05563042
## 222  0.05391053
## 223  0.05224558
## 224  0.05063397
## 225  0.04907332
## 226  0.04756238
## 227  0.04609917
## 228  0.0446822
## 229  0.04330759
## 230  0.04197593
## 231  0.04068634
## 232  0.03943768
## 233  0.03822812
## 234  0.0370564
## 235  0.03592146
## 236  0.03482136
## 237  0.03375633
## 238  0.0327246
## 239  0.03172487
## 240  0.03075593
## 241  0.02981687
## 242  0.02890735
## 243  0.02802622
## 244  0.02717285
## 245  0.02634558
## 246  0.02554371
```

```
## 247  0.02476675
## 248  0.0240143
## 249  0.02328515
## 250  0.02257836
## 251  0.02189306
## 252  0.02122923
## 253  0.02058575
## 254  0.01996238
## 255  0.01935788
## 256  0.01877213
## 257  0.01820429
## 258  0.01765402
## 259  0.01712051
## 260  0.01660351
## 261  0.01610242
## 262  0.0156171
## 263  0.01514632
## 264  0.01469009
## 265  0.01424752
## 266  0.01381863
## 267  0.01340292
## 268  0.01299987
## 269  0.01260905
## 270  0.01223013
## 271  0.01186274
## 272  0.01150682
## 273  0.0111617
## 274  0.01082683
## 275  0.01050223
## 276  0.01018802
## 277  0.009882836
## 278  0.009586995
## 279  0.009300183
## 280  0.009021979
## 281  0.008752229
## 282  0.008490788
## 283  0.008237117
## 284  0.007991165
## 285  0.007752791
## 286  0.007521485
## 287  0.00729729
## 288  0.0070798
## 289  0.006868829
## 290  0.006664461
## 291  0.006466233
## 292  0.006273943
```

```
## 293  0.006087383
## 294  0.005906438
## 295  0.005730878
## 296  0.005560733
## 297  0.005395614
## 298  0.005235502
## 299  0.005080266
## 300  0.004929713
## 301  0.004783633
## 302  0.004641939
## 303  0.004504489
## 304  0.004371175
## 305  0.004241788
## 306  0.004116328
## 307  0.003994633
## 308  0.003876575
## 309  0.003762146
## 310  0.003650998
## 311  0.003543299
## 312  0.003438761
## 313  0.003337309
## 314  0.003238867
## 315  0.0031434
## 316  0.003050815
## 317  0.002960989
## 318  0.002873863
## 319  0.002789256
## 320  0.002707202
## 321  0.002627575
## 322  0.002550329
## 323  0.002475383
## 324  0.002402639
## 325  0.002332083
## 326  0.002263665
## 327  0.002197253
## 328  0.0021328
## 329  0.002070281
## 330  0.002009582
## 331  0.001950708
## 332  0.001893577
## 333  0.001838136
## 334  0.001784306
## 335  0.001732112
## 336  0.001681432
## 337  0.0016323
## 338  0.001584614
```

```
## 339  0.001538318
## 340  0.00149336
## 341  0.001449755
## 342  0.001407453
## 343  0.001366422
## 344  0.001326558
## 345  0.001287868
## 346  0.001250318
## 347  0.001213876
## 348  0.00117851
## 349  0.0011442
## 350  0.001110891
## 351  0.001078564
## 352  0.001047205
## 353  0.001016748
## 354  0.000987183
## 355  0.0009584787
## 356  0.0009306203
## 357  0.0009035908
## 358  0.0008773552
## 359  0.0008518948
## 360  0.0008271908
## 361  0.0008032027
## 362  0.0007798997
## 363  0.0007572919
## 364  0.0007353517
## 365  0.0007140448
## 366  0.0006933782
## 367  0.0006733016
## 368  0.0006538091
## 369  0.0006348886
## 370  0.0006165295
## 371  0.0005986958
## 372  0.0005813859
## 373  0.0005645752
## 374  0.0005482732
## 375  0.0005324451
## 376  0.0005170752
## 377  0.00050215
## 378  0.0004876566
## 379  0.0004735905
## 380  0.0004599217
## 381  0.000446661
## 382  0.0004337932
## 383  0.000421295
## 384  0.000409168
```

```
## 385  0.000397383
## 386  0.0003859476
## 387  0.0003748359
## 388  0.000364068
## 389  0.0003535805
## 390  0.0003434142
## 391  0.0003335507
## 392  0.0003239612
## 393  0.0003146584
## 394  0.0003056151
## 395  0.0002968509
## 396  0.0002883257
## 397  0.0002800551
## 398  0.0002720189
## 399  0.000264218
## 400  0.0002566472
## 401  0.0002492882
## 402  0.0002421443
## 403  0.0002352033
## 404  0.000228468
## 405  0.000221927
## 406  0.0002155735
## 407  0.000209406
## 408  0.0002034151
## 409  0.0001976036
## 410  0.0001919538
## 411  0.000186465
## 412  0.000181138
## 413  0.0001759652
## 414  0.0001709464
## 415  0.0001660616
## 416  0.0001613203
## 417  0.0001567196
## 418  0.0001522457
## 419  0.0001479028
## 420  0.0001436849
## 421  0.0001395875
## 422  0.0001356086
## 423  0.0001317492
## 424  0.0001279981
## 425  0.0001243516
## 426  0.0001208117
## 427  0.0001173674
## 428  0.0001140324
## 429  0.0001107887
## 430  0.0001076371
```

```
## 431  0.000104577
## 432  0.0001016119
## 433  9.872603e-05
## 434  9.591727e-05
## 435  9.320263e-05
## 436  9.055879e-05
## 437  8.798708e-05
## 438  8.549038e-05
## 439  8.306588e-05
## 440  8.071403e-05
## 441  7.842657e-05
## 442  7.62079e-05
## 443  7.40515e-05
## 444  7.195051e-05
## 445  6.991543e-05
## 446  6.79373e-05
## 447  6.601425e-05
## 448  6.414913e-05
## 449  6.233487e-05
## 450  6.057473e-05
## 451  5.886107e-05
## 452  5.719856e-05
## 453  5.558319e-05
## 454  5.401246e-05
## 455  5.24875e-05
## 456  5.100614e-05
## 457  4.957083e-05
## 458  4.817137e-05
## 459  4.681194e-05
## 460  4.549512e-05
## 461  4.420892e-05
## 462  4.296327e-05
## 463  4.175395e-05
## 464  4.057565e-05
## 465  3.943508e-05
## 466  3.832371e-05
## 467  3.724535e-05
## 468  3.619579e-05
## 469  3.517841e-05
## 470  3.419127e-05
## 471  3.322741e-05
## 472  3.229243e-05
## 473  3.138653e-05
## 474  3.050298e-05
## 475  2.964781e-05
## 476  2.881304e-05
```

```
## 477  2.800681e-05
## 478  2.721956e-05
## 479  2.645602e-05
## 480  2.57129e-05
## 481  2.499174e-05
## 482  2.429051e-05
## 483  2.361012e-05
## 484  2.294818e-05
## 485  2.230642e-05
## 486  2.167962e-05
## 487  2.107264e-05
## 488  2.048336e-05
## 489  1.991114e-05
## 490  1.935353e-05
## 491  1.88103e-05
## 492  1.82841e-05
## 493  1.777284e-05
## 494  1.727619e-05
## 495  1.67922e-05
## 496  1.632309e-05
## 497  1.586889e-05
## 498  1.54238e-05
## 499  1.499398e-05
## 500  1.457493e-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

Datasets in PyTorch

Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2019). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.12.