rTorch + PyTorch

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

PyTorch with Rmarkdown

Chapter 1

Simple Regression with PyTorch

This examples combine Python and R code together.

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

1.1 Creating the network model

Our network model is a simple Linear layer with an input and an output shape of 1.

```
library(rTorch)
```

```
from __future__ import print_function

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable

torch.manual_seed(123)
#> <torch._C.Generator object at 0x7ffa385d2a90>
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

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

And the network output should be like this

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

1.1.1 Code in R

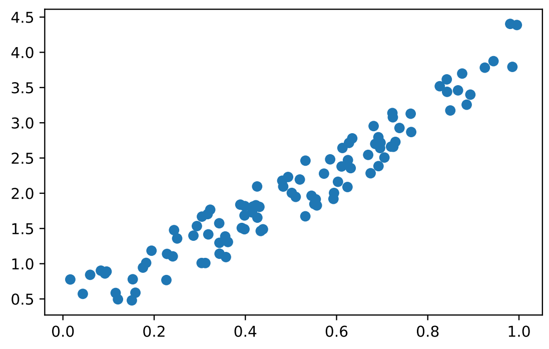
This would be the equivalent code in R:

```
library(reticulate)
#>
#> Attaching package: 'reticulate'
#> The following objects are masked from 'package:rTorch':
#>
#>     conda_install, conda_python

torch <- import("torch")
nn    <- import("torch.nn")
Variable <- import("torch.autograd")$Variable

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

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)
#> )
```

1.2 Datasets

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

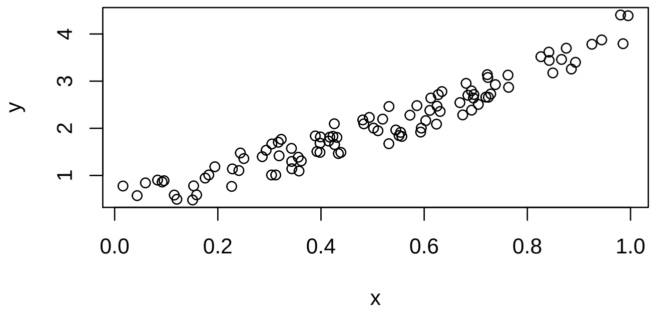
```
# Visualize our data
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(123)

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

plt.scatter(x, y)
plt.show()
```



This is the code in R:

```
np    <- import("numpy")

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

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

```
# convert numpy array to tensor in shape of input size
x = torch.from_numpy(x.reshape(-1,1)).float()
y = torch.from_numpy(y.reshape(-1,1)).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]]) tensor([[2.7166],
#>         [1.3983],
#>         [0.7679],
#>         [1.8464],
#>         [2.6614],
#>         [1.8297],
#>         [4.4034],
#>         [2.7003],
#>         [2.1778],
#>         [1.5073],
#>         [1.2966],
#>         [2.7287],
#>         [1.4884],
#>         [0.8423],
#>          [1.4895],
#>          [2.9263],
#>          [1.0114],
#>          [0.9445],
#>          [1.6729],
#>          [2.4624],
#>          [2.7788],
#>          [3.1746],
#>          [2.6593],
#>          [2.3800],
#>          [3.1382],
#>          [1.7665],
#>          [1.3082],
#>          [1.1390],
#>          [1.5341],
#>          [2.3566],
#>          [0.8612],
#>          [1.4642],
#>          [1.8066],
#>          [2.2308],
#>          [2.0962],
#>          [1.0096],
#>          [1.6538],
#>          [3.3994],
#>          [3.8747],
#>          [2.0045],
#>          [2.0884],
#>          [0.5845],
#>          [1.7039],
#>          [1.7285],
#>          [3.4602],
#>          [1.3581],
#>          [2.0949],
#>          [3.7935],
#>          [2.1950],
#>          [2.6425],
#>          [0.4948],
#>          [3.5188],
#>          [2.1628],
#>          [1.9643],
#>          [1.5740],
#>          [1.0099],
#>          [1.8123],
#>          [2.9534],
#>          [3.6986],
#>         [1.9485],
#>         [2.5445],
#>         [2.4811],
#>         [2.4700],
#>         [2.2838],
#>         [3.4392],
#>         [0.9015],
#>         [2.8687],
#>         [1.4766],
#>         [1.1847],
#>         [2.2782],
#>         [0.8885],
#>         [3.2565],
#>         [2.7141],
#>         [3.0781],
#>         [0.7763],
#>         [2.0038],
#>         [1.8270],
#>         [0.5882],
#>         [0.7793],
#>         [2.6416],
#>         [1.4162],
#>         [2.3851],
#>         [1.9140],
#>         [1.8385],
#>         [3.7822],
#>         [3.6160],
#>         [1.0941],
#>         [0.5721],
#>         [1.6683],
#>         [1.6848],
#>         [2.5068],
#>         [4.3876],
#>         [1.3866],
#>         [3.1286],
#>         [1.9197],
#>         [2.7949],
#>         [0.4797],
#>         [1.8171],
#>         [1.1042],
#>         [1.1414]])
```

1.2.1 Code in R

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]], dtype=torch.float64)
```

1.3 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()
```

1.3.1 Equivalent code in R

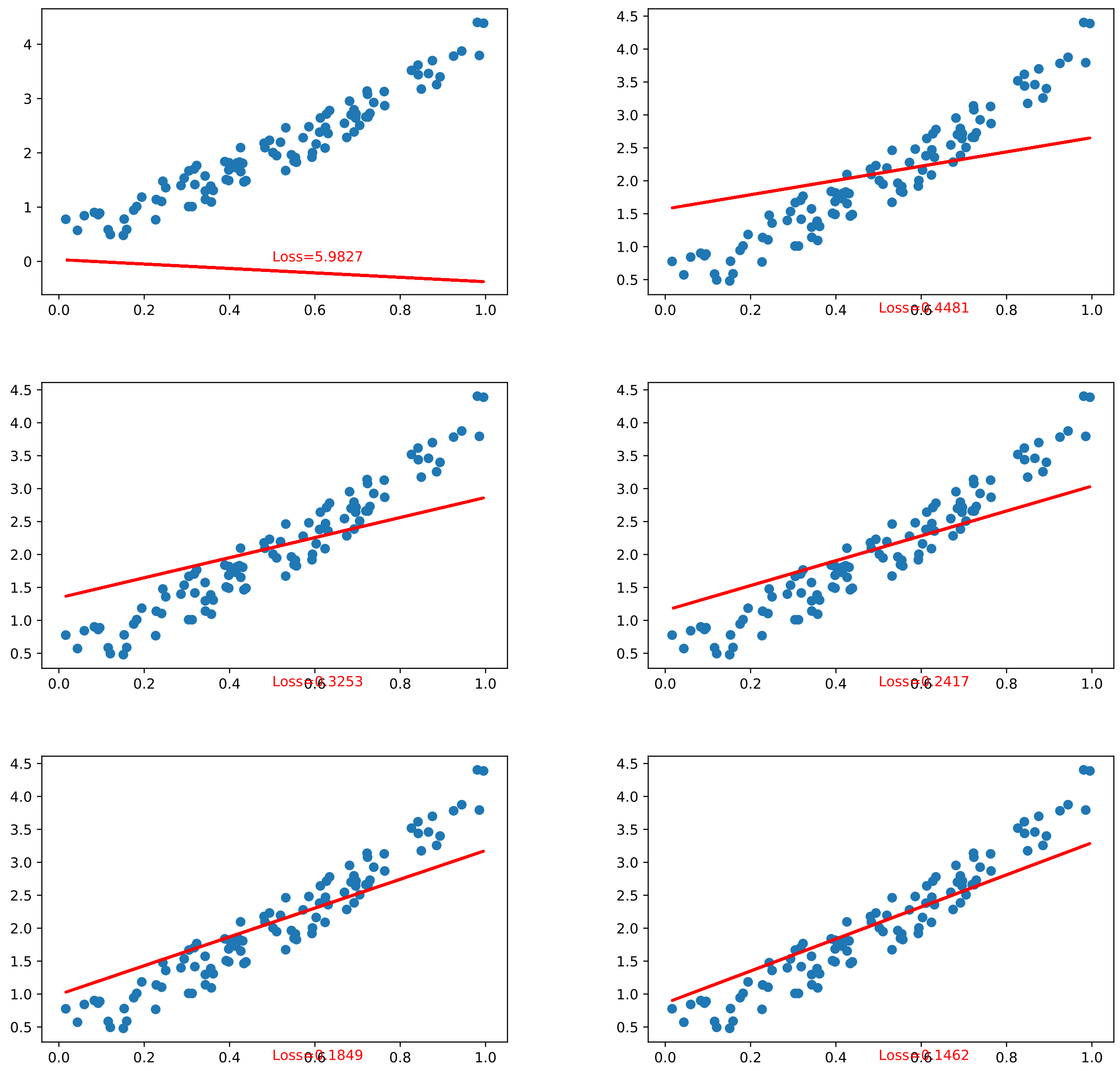
```
# Define Optimizer and Loss Function
optimizer <- torch$optim$SGD(net$parameters(), lr=0.2)
loss_func <- torch$nn$MSELoss()
print(optimizer)
#> SGD (
#> Parameter Group 0
#>     dampening: 0
#>     lr: 0.2
#>     momentum: 0
#>     nesterov: False
#>     weight_decay: 0
#> )
print(loss_func)
#> MSELoss()
```

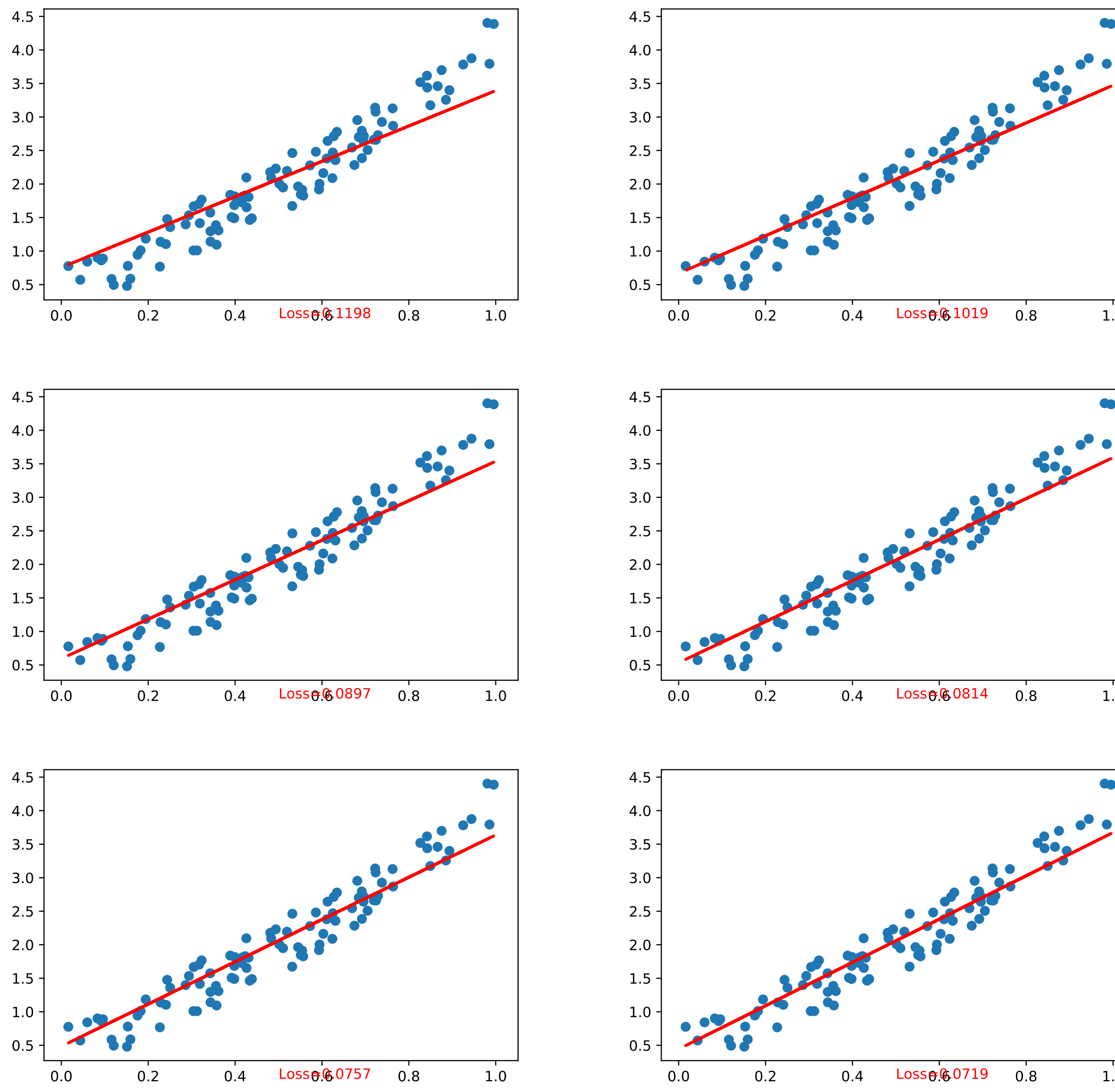
1.4 Training

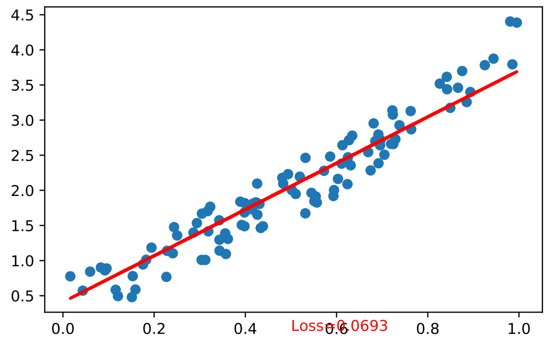
1.4.1 Code in Python

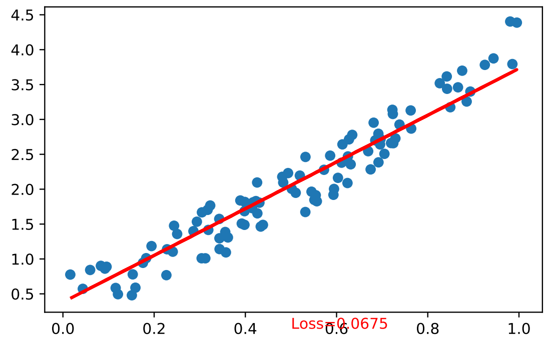
Now let's start our training process. With an epoch of 250, you will iterate our data to find the best value for our hyperparameters.

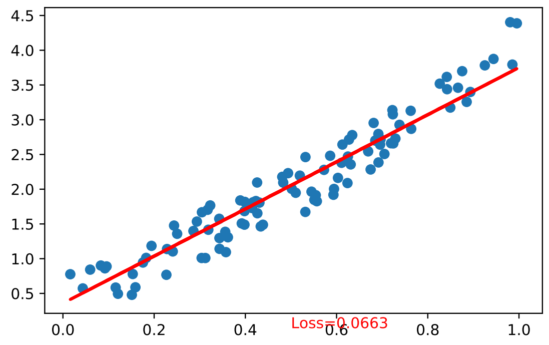
```
inputs = Variable(x)
outputs = Variable(y)
for i in range(250):
   prediction = net(inputs)
   loss = loss_func(prediction, outputs)
   optimizer.zero_grad()
   loss.backward()
   optimizer.step()

   if i % 10 == 0:
       # plot and show learning process
       plt.cla()
       plt.scatter(x.data.numpy(), y.data.numpy())
       plt.plot(x.data.numpy(), prediction.data.numpy(), 'r-', lw=2)
       plt.text(0.5, 0, 'Loss=%.4f' % loss.data.numpy(), fontdict={'size': 10, 'color'
       plt.pause(0.1)
```

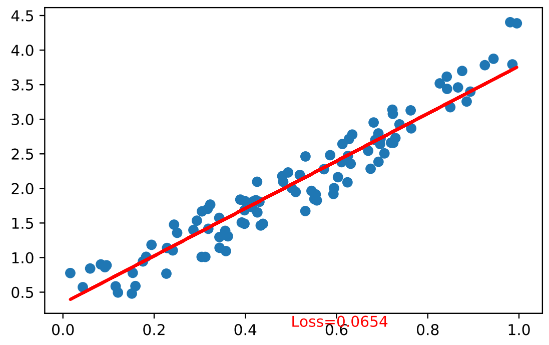

Loss=5.9827
Loss=0.4481
Loss=0.3253
Loss=0.2417
Loss=0.1849
Loss=0.1462

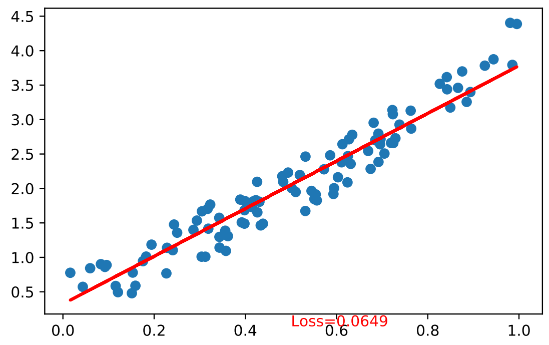

Loss=0.1198
Loss=0.1019
Loss=0.0897
Loss=0.0814
Loss=0.0757
Loss=0.0719

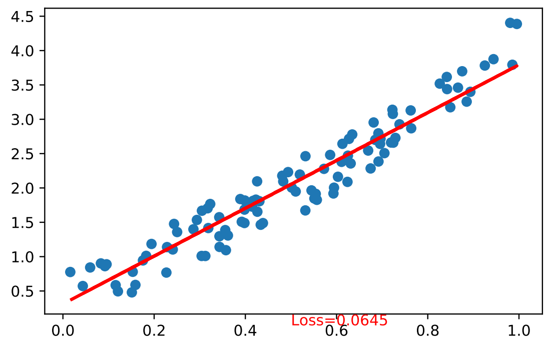

Loss=0.0693

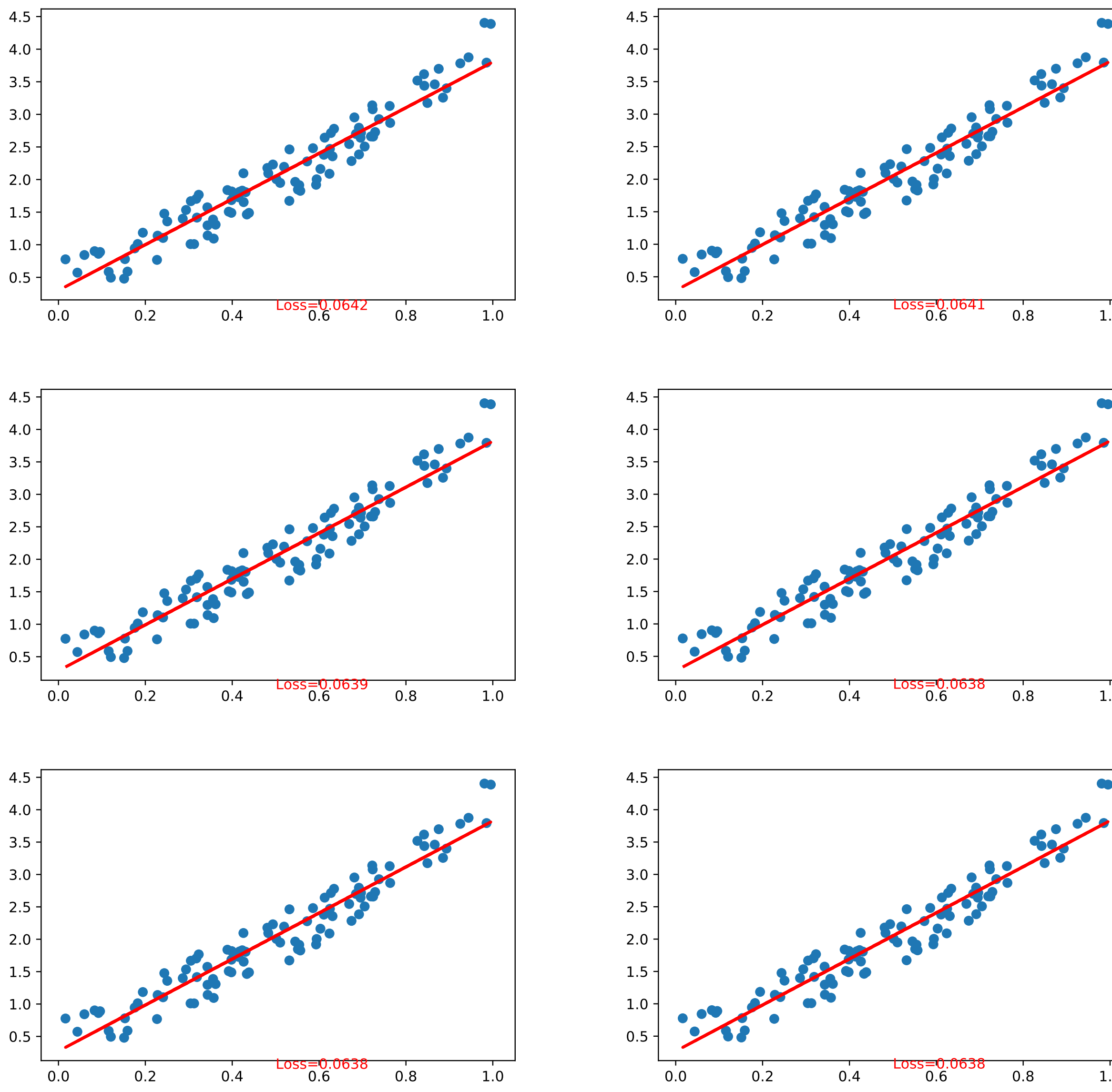

Loss=0.0675

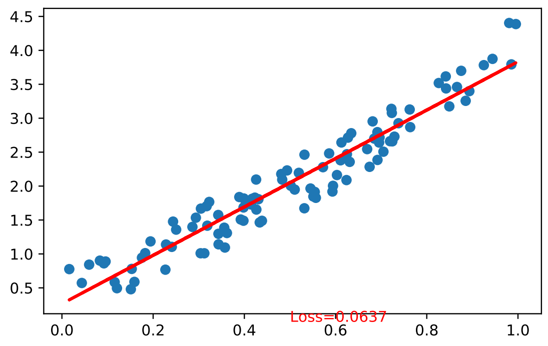

Loss=0.0663


Loss=0.0654


Loss=0.0649


Loss=0.0645

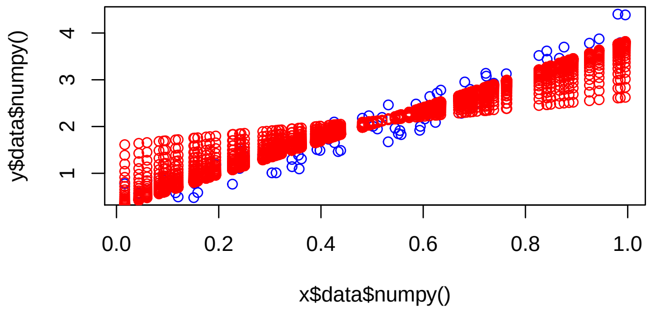

Loss=0.0642
Loss=0.0641
Loss=0.0639
Loss=0.0638
Loss=0.0638
Loss=0.0638



```
plt.show()
```

1.4.2 Code in R

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

inputs = Variable(x)
outputs = Variable(y)
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 %% 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")
   }
}
```



1.5 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

Chapter 2

Autograd

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

```r
library(rTorch)
```

2.1 Python code

```python
# Do not print from a function. Similar functionality to R invisible()
# https://stackoverflow.com/a/45669280/5270873
import os, sys

class HiddenPrints:
    def __enter__(self):
        self._original_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout.close()
        sys.stdout = self._original_stdout
```

```python
# Code in file autograd/two_layer_net_autograd.py
import torch
device = torch.device('cpu')
# device = torch.device('cuda') # Uncomment this to run on GPU

torch.manual_seed(0)

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
#> <torch._C.Generator object at 0x7f9444efdb90>
N, D_in, H, D_out = 64, 1000, 100, 10

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

# Create random Tensors for weights; setting requires_grad=True means that we
# want to compute gradients for these Tensors during the backward pass.
w1 = torch.randn(D_in, H, device=device, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, requires_grad=True)

learning_rate = 1e-6

for t in range(5):
  # Forward pass: compute predicted y using operations on Tensors. Since w1 and
  # w2 have requires_grad=True, operations involving these Tensors will cause
  # PyTorch to build a computational graph, allowing automatic computation of
  # gradients. Since we are no longer implementing the backward pass by hand we
  # don't need to keep references to intermediate values.
  y_pred = x.mm(w1).clamp(min=0).mm(w2)

  # Compute and print loss. Loss is a Tensor of shape (), and loss.item()
  # is a Python number giving its value.
  loss = (y_pred - y).pow(2).sum()
  print(t, loss.item())

  # Use autograd to compute the backward pass. This call will compute the
  # gradient of loss with respect to all Tensors with requires_grad=True.
  # After this call w1.grad and w2.grad will be Tensors holding the gradient
  # of the loss with respect to w1 and w2 respectively.
  loss.backward()

  # Update weights using gradient descent. For this step we just want to mutate
  # the values of w1 and w2 in-place; we don't want to build up a computational
  # graph for the update steps, so we use the torch.no_grad() context manager
  # to prevent PyTorch from building a computational graph for the updates
  with torch.no_grad():
    w1 -= learning_rate * w1.grad
    w2 -= learning_rate * w2.grad

    # Manually zero the gradients after running the backward pass
    with HiddenPrints():   # this would be the equivalent of invisible() in R
      w1.grad.zero_()
      w2.grad.zero_()
#> 0 29428666.0
#> 1 22739450.0
#> 2 20605262.0
#> 3 19520376.0
#> 4 17810228.0
```

2.2 R code

```
# library(reticulate) # originally qwe used reticulate
library(rTorch)

torch  = import("torch")
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.
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)

# Create random Tensors for weights; setting requires_grad=True means that we
# want to compute gradients for these Tensors during the backward pass.
w1 = torch$randn(D_in, H, device=device, requires_grad=TRUE)
w2 = torch$randn(H, D_out, device=device, requires_grad=TRUE)

learning_rate = torch$scalar_tensor(1e-6)

for (t in 1:5) {
  # Forward pass: compute predicted y using operations on Tensors. Since w1 and
  # w2 have requires_grad=True, operations involving these Tensors will cause
  # PyTorch to build a computational graph, allowing automatic computation of
  # gradients. Since we are no longer implementing the backward pass by hand we
  # don't need to keep references to intermediate values.
  y_pred = x$mm(w1)$clamp(min=0)$mm(w2)

  # Compute and print loss. Loss is a Tensor of shape (), and loss.item()
  # is a Python number giving its value.
  loss = (torch$sub(y_pred, y))$pow(2)$sum()
  cat(t, "\t", loss$item(), "\n")

  # Use autograd to compute the backward pass. This call will compute the
  # gradient of loss with respect to all Tensors with requires_grad=True.
  # After this call w1.grad and w2.grad will be Tensors holding the gradient
  # of the loss with respect to w1 and w2 respectively.
  loss$backward()

  # Update weights using gradient descent. For this step we just want to mutate
  # the values of w1 and w2 in-place; we don't want to build up a computational
  # graph for the update steps, so we use the torch.no_grad() context manager
  # to prevent PyTorch from building a computational graph for the updates
  with(torch$no_grad(), {
    w1$data = torch$sub(w1$data, torch$mul(w1$grad, learning_rate))
    w2$data = torch$sub(w2$data, torch$mul(w2$grad, learning_rate))

    # Manually zero the gradients after running the backward pass
    w1$grad$zero_()
    w2$grad$zero_()
  })
}
#> 1     29428666
#> 2     22739450
#> 3     20605262
#> 4     19520376
#> 5     17810228
```

2.3 Observations

If the seeds worked the same in Python and R, we should see similar results in the output.

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 25th percentile, second quantile AKA median AKA 50th percentile, third quantile AKA 75th, 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.