# Introduction to Machine Learning Implementation with PyTorch

*Most of the basic ideas are from 모두를 위한 머신러닝/딥러닝 강의 (http://hunkim.github.io/ml/). All implementations posted can only be used for educational purposes with references to the original author, Sangjun Park. MS, Korea University, College of Medicine*

## Brief overview of PyTorch

PyTorch is an open-source python package explicitly developed for advanced research and application on deep learning. Pros when compared to Tensorflow are wide accessibility, intuitiveness of design, and **extreme dynamicity** (which is the most fundamental difference between the two modules) when dealing with computation graphs. The only disadvantage regarding PyTorch is that compared to its superiority, it is still relatively a more recent product than Tensorflow, which makes it harder to gain much attention in most corporate settings. However, it is certain that the current paradigm is being shifted towards what we will discuss further, as more and more researchers are applying PyTorch as the main source of their works.

Installation, basic concepts and syntax documentations can be found at the official pytorch website (https://pytorch.org/).

## Linear Regression

### Single-variable Linear Regression

The mathematical model for single-variable linear regression is as follows.

$$y = \beta_0 + \beta_1 x$$

We call $\beta_1$ as the regression coefficient or the slope, and $\beta_0$ as the intercept. This model is also called simple linear regression.

Let us first examine the exact solution using the least squares method. The method of least squares states that the sum of squares between the actual value and the fitted value, should be minimum. Using formal notations, minimization of

$$\Big( \underbrace{\hat{y}}_{\text{Fitted value}} - \underbrace{y}_{\text{Actual value}} \Big)^2$$

is the goal of this model.

The exact solution to this problem is the following whose derivations will be omitted.

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \; (\bar{x},\, \bar{y} = Mean\ values\ of\ x\ and\ y)$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

For this example, we will use the following data.

$$x = \begin{bmatrix} 1.05 & 1.54 & 2.01 & 2.55 & 3.13 & 3.45 & 4.02 & 4.65 & 5.34 & 5.50 \end{bmatrix}^T$$
$$y = \begin{bmatrix} 10.3 & 15.4 & 21.2 & 25.4 & 30.1 & 35.6 & 40.5 & 43.4 & 50.1 & 56.6 \end{bmatrix}^T$$

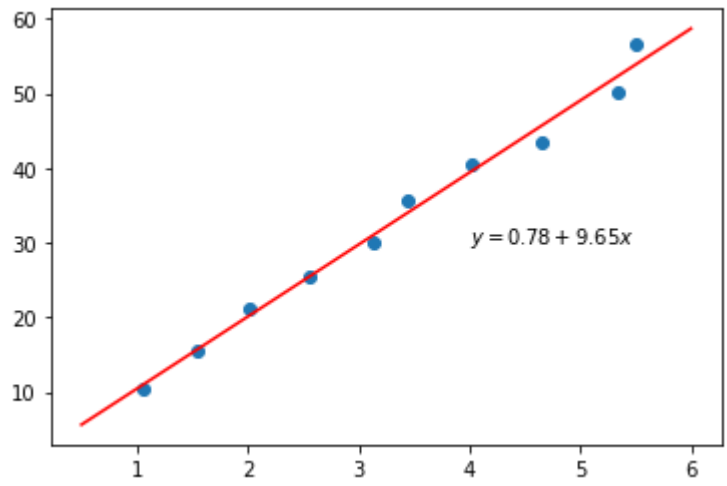
In [33]:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([1.05, 1.54, 2.01, 2.55, 3.13, 3.45, 4.02, 4.65, 5.34, 5.50])
y = np.array([10.3, 15.4, 21.2, 25.4, 30.1, 35.6, 40.5, 43.4, 50.1, 56.6])

b1 = np.sum((x - np.mean(x)) * (y - np.mean(y))) / np.sum((x - np.mean(x))**2)
b0 = np.mean(y) - b1 * np.mean(x)

x_range = np.linspace(0.5, 6.0, 1000)
y_hat = b0 + b1 * x_range

print(b0, b1)

plt.scatter(x, y)
plt.plot(x_range, y_hat, 'r')
plt.text(4, 30, "$y = 0.78 + 9.65x$")
```

```
0.7824239351471718 9.650293641652478
```

Out[33]:

```
Text(4, 30, '$y = 0.78 + 9.65x$')
```




Next, let's use the gradient descent method using PyTorch.

In [119]:

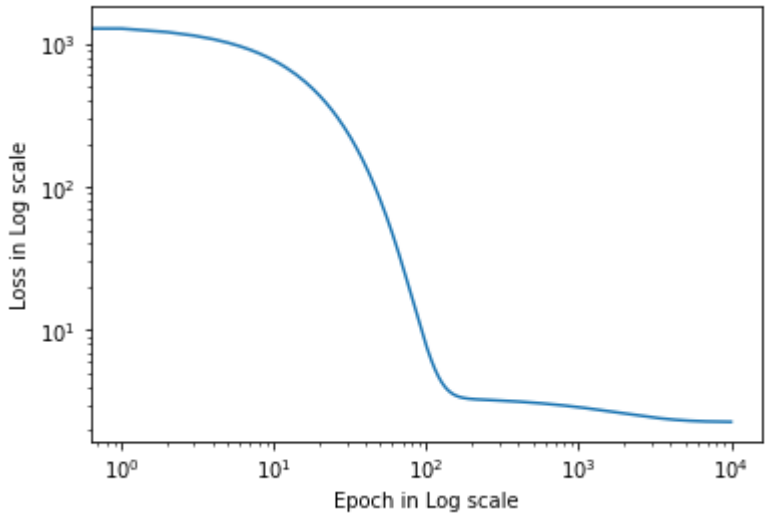
```
import torch
from torch.autograd import Variable

class linearRegression(torch.nn.Module):
    def __init__(self, inputSize, outputSize):
        super(linearRegression, self).__init__()
        self.linear = torch.nn.Linear(inputSize, outputSize, bias = True)

    def forward(self, x):
        out = self.linear(x)
        return out

inputDim = 1
outputDim = 1
learningRate = 0.001
epochs = 10000

model = linearRegression(inputDim, outputDim)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learningRate)

x_torch = torch.from_numpy(x).float().unsqueeze(dim=1)
y_torch = torch.from_numpy(y).float().unsqueeze(dim=1)

epoch_graph = np.array([])
loss_graph = np.array([])

for epoch in range(epochs):

    inputs = x_torch
    labels = y_torch

    # Clear gradient buffers because we don't want any gradient from previous epoch to carry for
ward, dont want to cumulate gradients
    optimizer.zero_grad()

    # get output from the model with given inputs
    outputs = model(inputs)

    # get loss for the predicted output
    loss = criterion(outputs, labels)

    # get gradients with respect to parameters
    loss.backward()

    # update parameters
    optimizer.step()

    epoch_graph = np.append(epoch_graph, epoch)
    loss_graph = np.append(loss_graph, loss.item())
```

In [115]:

```
plt.loglog(epoch_graph, loss_graph)
plt.xlabel('Epoch in Log scale')
plt.ylabel('Loss in Log scale')
```

Out[115]:

```
Text(0, 0.5, 'Loss in Log scale')
```




In [116]:

```
for param in model.parameters():
    print(param)
```

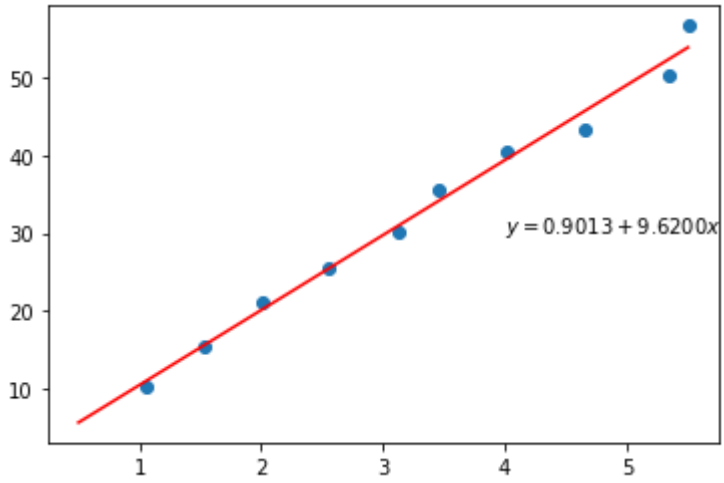
```
Parameter containing:
tensor([[9.6200]], requires_grad=True)
Parameter containing:
tensor([0.9013], requires_grad=True)
```

In [117]:

```
x_range2 = np.linspace(0.5, 5.5, 10)
y_hat2 = model(torch.from_numpy(x_range2).float().unsqueeze(dim=1))
plt.scatter(x, y)
plt.plot(x_range2, y_hat2.detach().numpy(), 'r')
plt.text(4, 30, "$y=0.9013+9.6200x$")
```

Out[117]:

Text(4, 30, '$y=0.9013+9.6200x$')




**Multiple Linear Regression**

The model is the following

$$Y = X\beta$$

where

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \ddots & & & \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}$$

and $n$ is the number of observations, and $p$ is the number of dependent variables.

Let us derive the exact solution via the least squares method and matrix algebra. The residual term would be

$$e = Y - X\beta$$

which makes the least squares or the loss

$$\begin{aligned} e^T e &= (Y - X\beta)^T (Y - X\beta) \\ &= Y^T Y - \beta^T X^T Y - Y^T X\beta + \beta^T X^T X\beta \end{aligned}$$

Thus, differentiating the least square value with respect to $\beta$ should be zero which leads to the following result.

$$\begin{aligned} \frac{\partial (e^T e)}{\partial \beta} &= -2X^T y + 2X^T X\beta \\ &= 0 \end{aligned}$$

The calculated regression coefficient $\beta$ would be

$$\therefore \beta = (X^T X)^{-1} X^T Y$$

Using numpy, we can get the exact solution. For this example, let us use the test scores of general psychology open to the public.

In [146]:

```
import pandas as pd

link = 'https://college.cengage.com/mathematics/brase/understandable_statistics/7e/students/datasets/mlr/excel/mlr03.xls'
df = pd.read_excel(link)
df
```

*** No CODEPAGE record, no encoding_override: will use 'ascii'

Out[146]:

| | EXAM1 | EXAM2 | EXAM3 | FINAL |
|---|---|---|---|---|
| **0** | 73 | 80 | 75 | 152 |
| **1** | 93 | 88 | 93 | 185 |
| **2** | 89 | 91 | 90 | 180 |
| **3** | 96 | 98 | 100 | 196 |
| **4** | 73 | 66 | 70 | 142 |
| **5** | 53 | 46 | 55 | 101 |
| **6** | 69 | 74 | 77 | 149 |
| **7** | 47 | 56 | 60 | 115 |
| **8** | 87 | 79 | 90 | 175 |
| **9** | 79 | 70 | 88 | 164 |
| **10** | 69 | 70 | 73 | 141 |
| **11** | 70 | 65 | 74 | 141 |
| **12** | 93 | 95 | 91 | 184 |
| **13** | 79 | 80 | 73 | 152 |
| **14** | 70 | 73 | 78 | 148 |
| **15** | 93 | 89 | 96 | 192 |
| **16** | 78 | 75 | 68 | 147 |
| **17** | 81 | 90 | 93 | 183 |
| **18** | 88 | 92 | 86 | 177 |
| **19** | 78 | 83 | 77 | 159 |
| **20** | 82 | 86 | 90 | 177 |
| **21** | 86 | 82 | 89 | 175 |
| **22** | 78 | 83 | 85 | 175 |
| **23** | 76 | 83 | 71 | 149 |
| **24** | 96 | 93 | 95 | 192 |

In [176]:

```
data = np.array(df.values)
data
```

Out[176]:

```
array([[ 73,  80,  75, 152],
       [ 93,  88,  93, 185],
       [ 89,  91,  90, 180],
       [ 96,  98, 100, 196],
       [ 73,  66,  70, 142],
       [ 53,  46,  55, 101],
       [ 69,  74,  77, 149],
       [ 47,  56,  60, 115],
       [ 87,  79,  90, 175],
       [ 79,  70,  88, 164],
       [ 69,  70,  73, 141],
       [ 70,  65,  74, 141],
       [ 93,  95,  91, 184],
       [ 79,  80,  73, 152],
       [ 70,  73,  78, 148],
       [ 93,  89,  96, 192],
       [ 78,  75,  68, 147],
       [ 81,  90,  93, 183],
       [ 88,  92,  86, 177],
       [ 78,  83,  77, 159],
       [ 82,  86,  90, 177],
       [ 86,  82,  89, 175],
       [ 78,  83,  85, 175],
       [ 76,  83,  71, 149],
       [ 96,  93,  95, 192]], dtype=int64)
```

In [232]:

```
one = np.ones((25, 1))
x = np.concatenate((one, data[:, 0:3]), axis=1)
y = np.array(data[:, 3])

beta = np.linalg.inv(np.transpose(x) @ x) @ np.transpose(x) @ y
```

In [233]:

```
beta
```

Out[233]:

```
array([-4.3361024 ,  0.35593822,  0.54251876,  1.16744422])
```

Now, let's use PyTorch.

In [247]:

```
x = np.array(data[:, 0:3])

import torch
from torch.autograd import Variable

class MultivariateLinearRegressionModel(torch.nn.Module):
    def __init__(self, inputSize, outputSize):
        super(MultivariateLinearRegressionModel, self).__init__()
        self.linear = torch.nn.Linear(inputSize, outputSize, bias = True)

    def forward(self, x):
        out = self.linear(x)
        return out

inputDim = 3
outputDim = 1
learningRate = 0.00005
epochs = 50000

model = MultivariateLinearRegressionModel(inputDim, outputDim)
criterion = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learningRate)

x_torch = torch.from_numpy(x).float()
y_torch = torch.from_numpy(y).float().unsqueeze(dim=1)

epoch_graph = np.array([])
loss_graph = np.array([])

for epoch in range(epochs):
    inputs = x_torch
    labels = y_torch

    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    epoch_graph = np.append(epoch_graph, epoch)
    loss_graph = np.append(loss_graph, loss.item())

    if epoch % 1000 == 0:
        print("Epoch {ep}: Loss = {lo}".format(ep=epoch, lo=loss.item()))
```

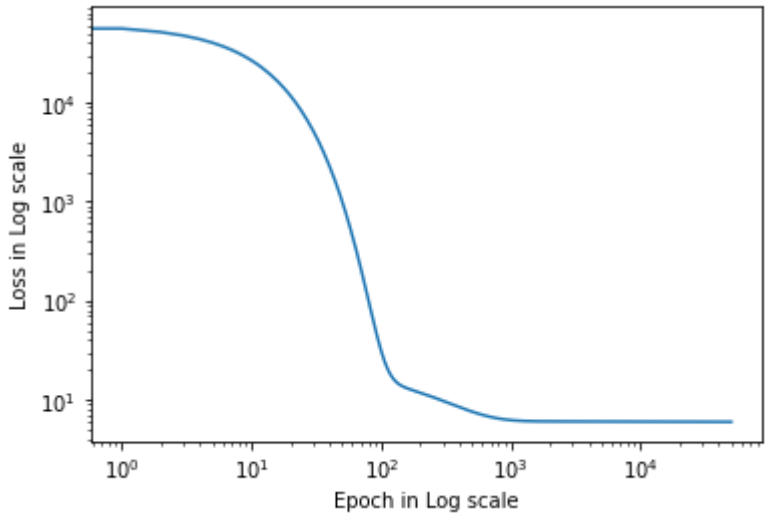
```
Epoch 0: Loss = 60733.30078125
Epoch 1000: Loss = 6.238600254058838
Epoch 2000: Loss = 6.054095268249512
Epoch 3000: Loss = 6.048786640167236
Epoch 4000: Loss = 6.047374248504639
Epoch 5000: Loss = 6.0461649894714355
Epoch 6000: Loss = 6.044985294342041
Epoch 7000: Loss = 6.043803691864014
Epoch 8000: Loss = 6.042622089385986
Epoch 9000: Loss = 6.041445255279541
Epoch 10000: Loss = 6.040285587310791
Epoch 11000: Loss = 6.039116859436035
Epoch 12000: Loss = 6.037960052490234
Epoch 13000: Loss = 6.036801815032959
Epoch 14000: Loss = 6.035653591156006
Epoch 15000: Loss = 6.03449821472168
Epoch 16000: Loss = 6.033360481262207
Epoch 17000: Loss = 6.032228946685791
Epoch 18000: Loss = 6.031095027923584
Epoch 19000: Loss = 6.029968738555908
Epoch 20000: Loss = 6.028834342956543
Epoch 21000: Loss = 6.027714252471924
Epoch 22000: Loss = 6.026602268218994
Epoch 23000: Loss = 6.025495529174805
Epoch 24000: Loss = 6.02438497543335
Epoch 25000: Loss = 6.023271083831787
Epoch 26000: Loss = 6.0221943855285645
Epoch 27000: Loss = 6.021090507507324
Epoch 28000: Loss = 6.019993305206299
Epoch 29000: Loss = 6.0189080238342285
Epoch 30000: Loss = 6.017822265625
Epoch 31000: Loss = 6.016748428344727
Epoch 32000: Loss = 6.015677452087402
Epoch 33000: Loss = 6.014606952667236
Epoch 34000: Loss = 6.013540267944336
Epoch 35000: Loss = 6.012478828430176
Epoch 36000: Loss = 6.0114216804504395
Epoch 37000: Loss = 6.010371685028076
Epoch 38000: Loss = 6.009322643280029
Epoch 39000: Loss = 6.008278846740723
Epoch 40000: Loss = 6.007232666015625
Epoch 41000: Loss = 6.00619649887085
Epoch 42000: Loss = 6.005164623260498
Epoch 43000: Loss = 6.004148483276367
Epoch 44000: Loss = 6.003118991851807
Epoch 45000: Loss = 6.002092361450195
Epoch 46000: Loss = 6.001070499420166
Epoch 47000: Loss = 6.000068187713623
Epoch 48000: Loss = 5.99906063079834
Epoch 49000: Loss = 5.998051166534424
```

In [248]:

```
plt.loglog(epoch_graph, loss_graph)
plt.xlabel('Epoch in Log scale')
plt.ylabel('Loss in Log scale')
```

Out[248]:

```
Text(0, 0.5, 'Loss in Log scale')
```




In [249]:

```
for params in model.parameters():
    print(params)
```

```
Parameter containing:
tensor([[0.3560, 0.5322, 1.1333]], requires_grad=True)
Parameter containing:
tensor([-0.6695], requires_grad=True)
```

In [ ]:

```

```