國立高雄科技大學
National Kaohsiung University of Science and Technology

Beginner's Guide on PyTorch from Linear Regression

Speaker: Shih-Shinh Huang

Version: v011

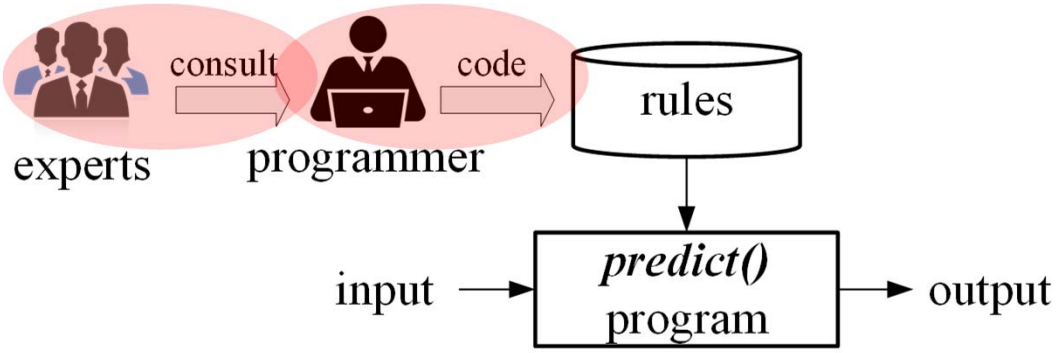
Date: October 8, 2020

Outline

- Introduction
  - About Machine Learning
  - Basic Learning Paradigm
  - Support from PyTorch
- Linear Regression
  - Problem Description
  - Gradient Descent Algorithm
  - Numpy Program
- PyTorch Programming
  - Autograd and Optimizer
  - Data Providing
  - Model Building

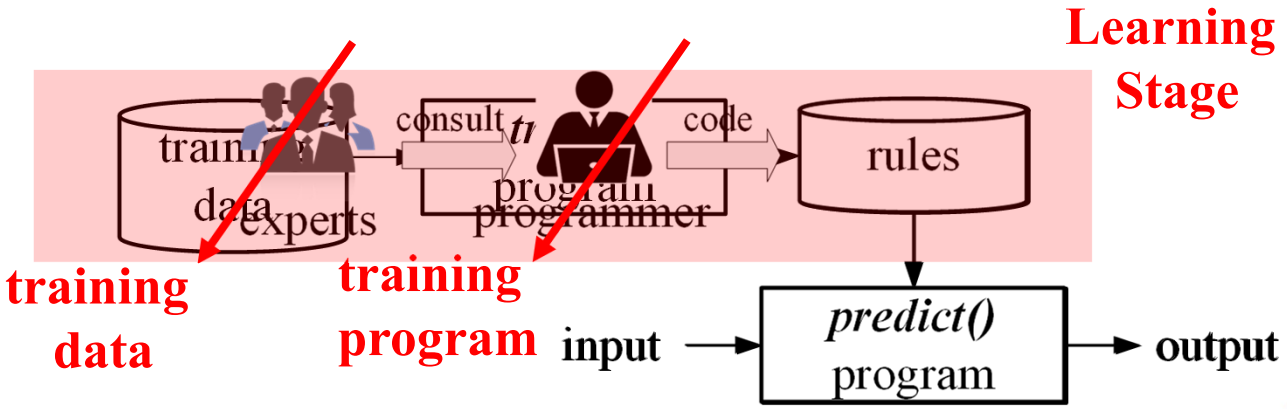
Introduction

- About Machine Learning (ML)
  - Traditional programming is to code the rules by the programmer
    - the rules is for producing output
    - the rules are obtained by consulting domain experts



Introduction

- About Machine Learning (ML)
  - ML is to learn the rules (relations between input and output) from training data
    - programmer → training program
    - experts → data

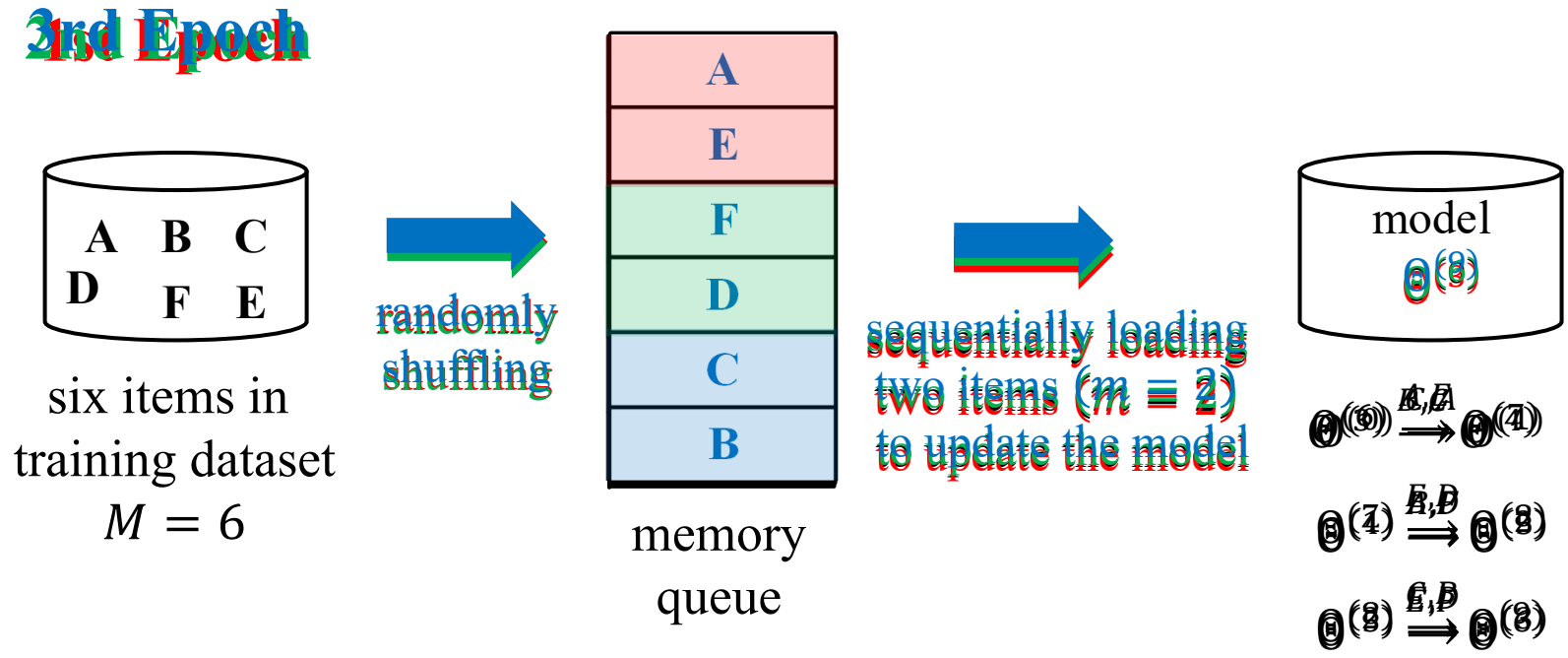


Introduction

- Basic Learning Paradigm
  - Step 1: randomly initialize model to be trained
  - Step 2: repeatedly visit all items in the training dataset (with M data items) for N times (epochs)
    - Step 2.1: randomly shuffle the data items
    - Step 2.2: sequentially load m data items (mini-batch) to update the model (totally, M/m updates)

Introduction

- Example: $(M = 6, m = 2, N = 3)$

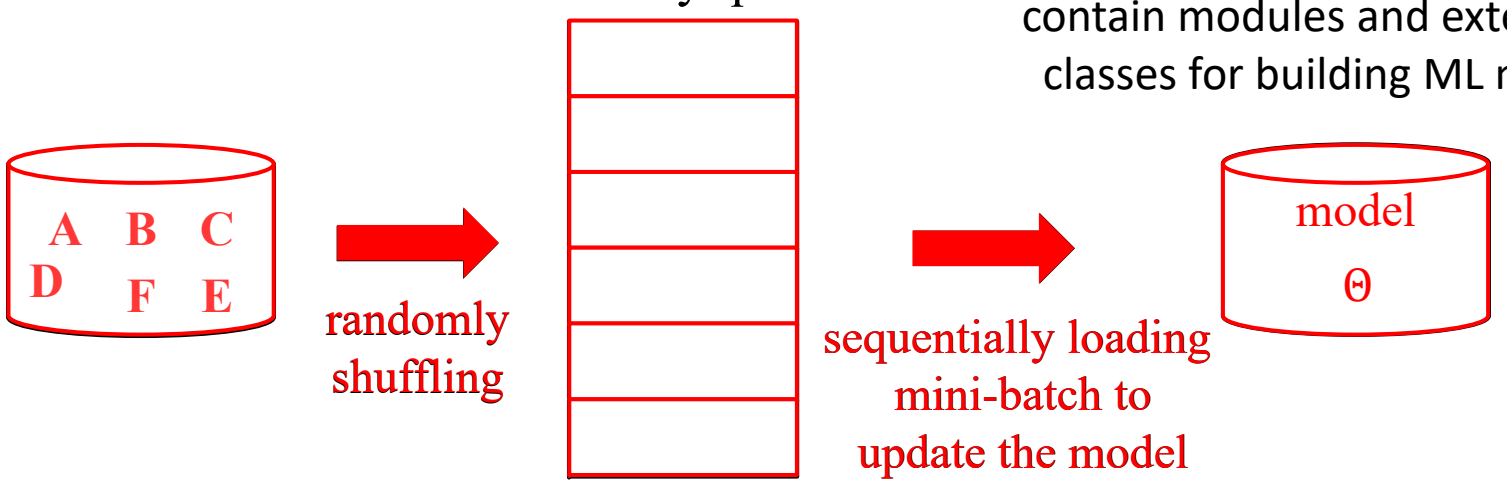


Introduction

- Support from PyTorch
  - PyTorch was developed by Facebook and is an machine learning package for Python language
    - support dataset management and exploring
    - support model building
    - support optimization utilities
    - support parallel processing by using GPU

Introduction

and exploring



torch.nn subpackage

contain modules and extensible classes for building ML model

torch package

contain library for the defined tensor data type, similar to Numpy ndarray but use GPU

torch.utils.data subpackage

contain dataset and loaders to make data processing easier

torch.autograd s package

provide

torch.

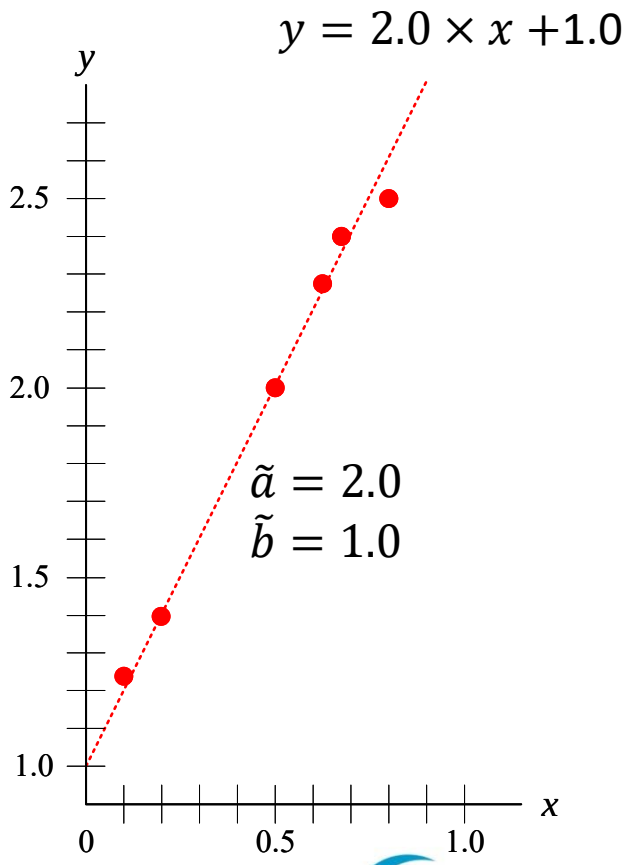
contain s

operatio

Linear Regression

- Problem Description
  - Given: N points $\{(x_i, y_i)\}_{i=1}^{N}$ sampling from a linear model $y_i = ax_i + b$ with noise
  - Objective: find the model parameters $(\tilde{a}, \tilde{b})$ that most fits these points (minimize fitting error)

i	1	2	3	4	5	6
x_i	0.11	0.20	0.49	0.65	0.67	0.83
y_i	1.24	1.40	2.00	2.27	2.41	2.52



Linear Regression

- Problem Description
  - Mean Square Error (MSE) (or called loss) is a way for measuring fitting error

$$L_{MSE}(a,b) = \frac{1}{N} \times \sum_{i} (y_i - \hat{y}_i)^2$$

ground truth

predicted value

$$\hat{y}_i = a \times x_i + b$$

x_i	0.11	0.20	0.49	0.65	0.67	0.83
y_i	1.24	1.40	2.00	2.27	2.41	2.52

$(a, b) = (1.0, 1.0)$ $\quad \hat{y}_i = 1.0 \times x_i + 1.0$

$\hat{y}_i$	1.11	1.20	1.49	1.65	1.67	1.83
$y_i - \hat{y}_i$	0.13	0.20	0.51	0.62	0.74	0.69

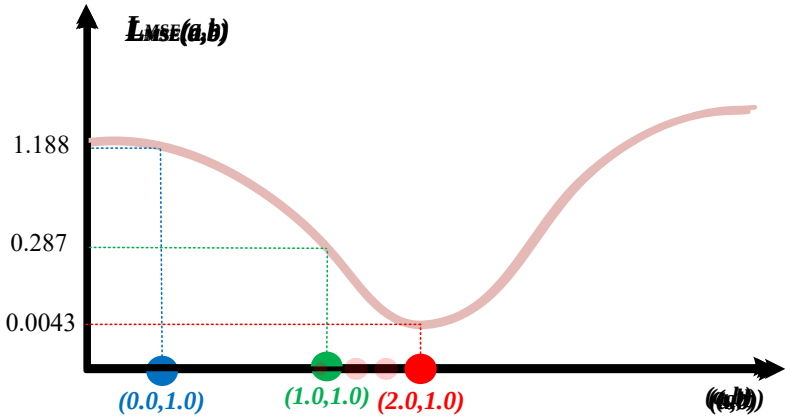
$$L_{MSE}(a = 1.0, b = 1.0)$$

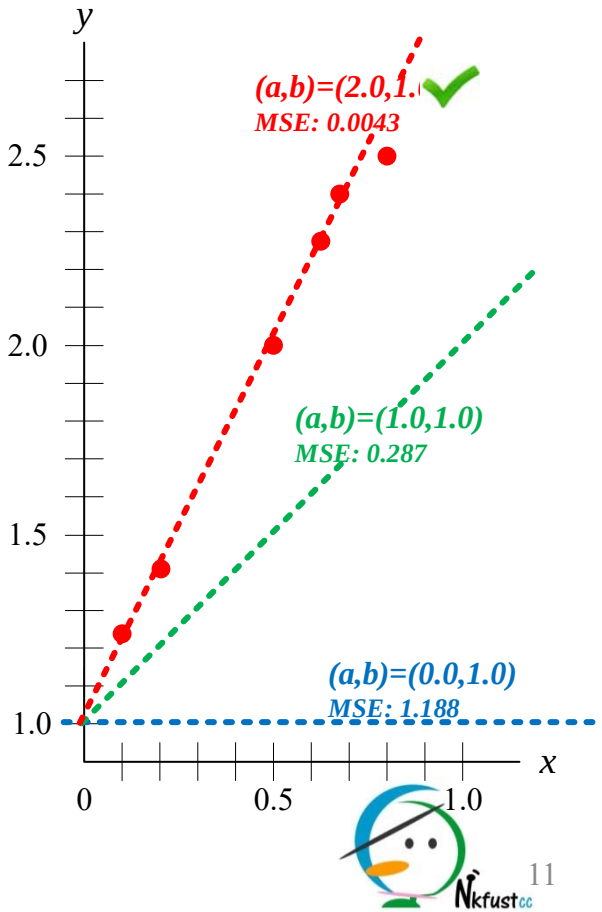
$$= \frac{1}{6}(0.13^2 + 0.20^2 + 0.51^2 + 0.62^2 + 0.74^2 + 0.69^2)$$

Linear Regression

- Problem Description
  - $(\tilde{a}, \tilde{b})$ is the model parameters that minimizes $L_{MSE}(a, b)$

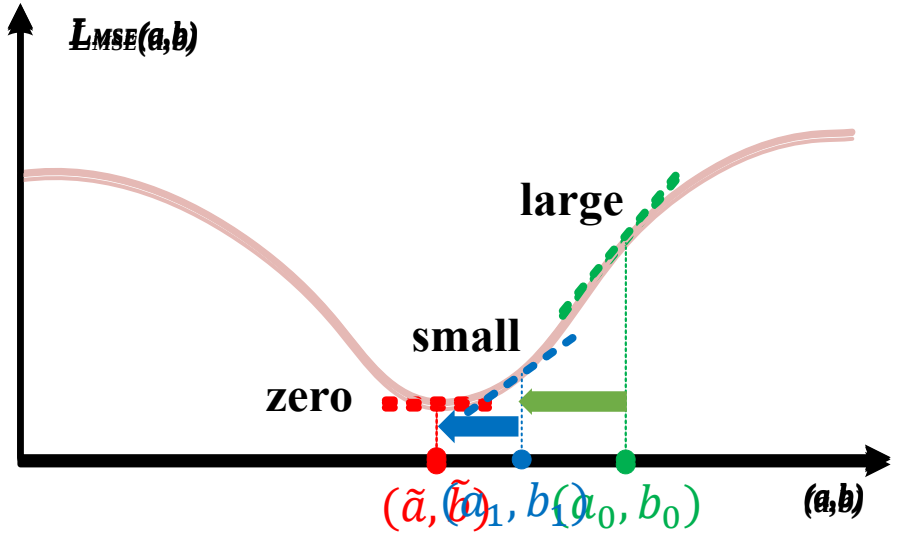
$$(\tilde{a}, \tilde{b}) = \arg\min_{(a,b)} L_{MSE}(a, b)$$





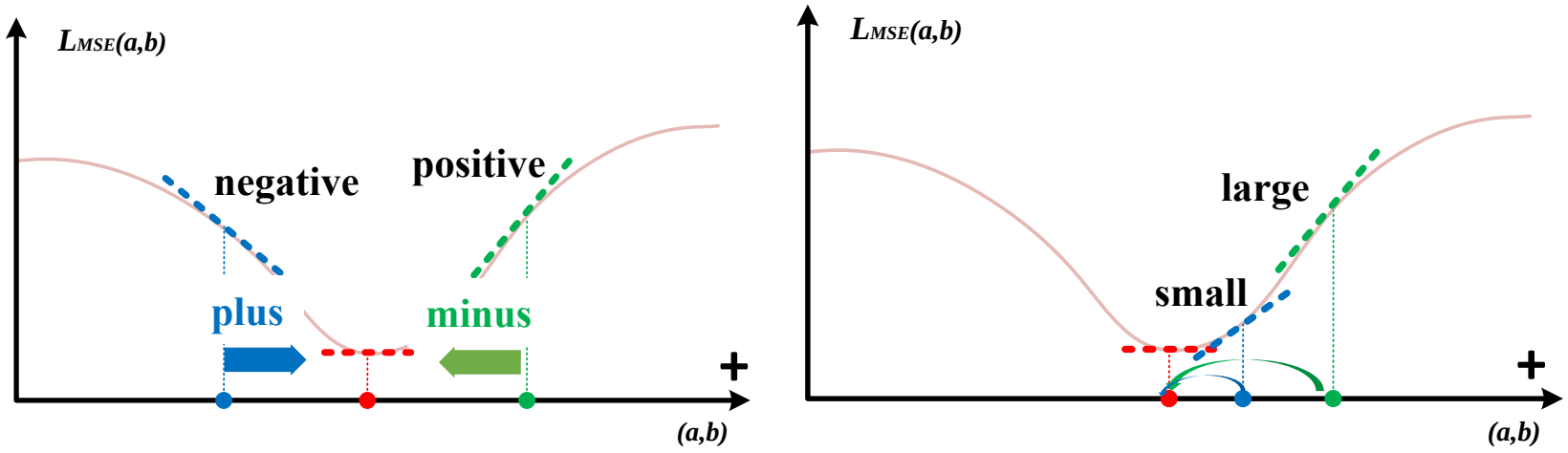
Linear Regression

- Gradient Descent Algorithm
  - minimize MSE loss in an iterative manner
    - start from a randomly selected point (a_0, b_0)
    - move toward the minimal point with zero gradient by gradually decreasing the gradient



Linear Regression

- Gradient Descent Algorithm
  - take a step proportional to the negative direction of gradient
    - negative→ increase (+); positive → decrease(-);
    - large → far → big step; small → near → small step



Linear Regression

- Gradient Descent Algorithm
  - take a step proportional to the negative direction of gradient

$$a_{t+1} = a_t + \eta\alpha_t \frac{\partial L_{MSE}(a_t, b_t)}{\partial a} \propto \frac{\partial L_{MSE}(a_t, b_t)}{\partial a}$$

$$b_{t+1} = b_t + \eta b_t \frac{\partial L_{MSE}(a_t, b_t)}{\partial b} \propto \frac{\partial L_{MSE}(a_t, b_t)}{\partial b}$$

  - η: learning rate

$$\frac{\partial L_{MSE}(a,b)}{\partial a} = \frac{1}{\partial a}\left(\frac{1}{N}\times\sum_i\left(y_i-(ax_i+b)\right)^2\right)$$

$$=\frac{1}{N}\times\sum_i\frac{1}{\partial a}\left(y_i-(ax_i+b)\right)^2$$

$$=\frac{1}{N}\times\sum_i 2\times\left(y_i-(ax_i+b)\right)\times\frac{1}{\partial a}\left(y_i-(ax_i+b)\right)$$

$$=\frac{1}{N}\times\sum_i 2\times\left(y_i-(ax_i+b)\right)\times(-x_i)$$

$$\frac{\partial L_{MSE}(a,b)}{\partial b} = \frac{1}{\partial b}\left(\frac{1}{N}\times\sum_i\left(y_i-(ax_i+b)\right)^2\right)$$

$$=\frac{1}{N}\times\sum_i\frac{1}{\partial b}\left(y_i-(ax_i+b)\right)^2$$

$$=\frac{1}{N}\times\sum_i 2\times\left(y_i-(ax_i+b)\right)\times\frac{1}{\partial b}\left(y_i-(ax_i+b)\right)$$

$$=\frac{1}{N}\times\sum_i 2\times\left(y_i-(ax_i+b)\right)\times(-1)$$

Linear Regression

- Gradient Descent Algorithm

$$a_{t+1} = a_t - \eta \times \frac{\partial L_{MSE}(a_t, b_t)}{\partial a}$$

$$= a_t - \eta \times \left(-2 \times \frac{1}{N} \times \sum_i (y_i - (a_t x_i + b_t)) \times x_i \right)$$

$$b_{t+1} = b_t - \eta \times \frac{\partial L_{MSE}(a_t, b_t)}{\partial b}$$

$$= b_t - \eta \times \left(-2 \times \frac{1}{N} \times \sum_i (y_i - (a_t x_i + b_t)) \right)$$

predicted value at time t

$\hat{y}_{i,t} = a_t \times x_i + b_t$

Linear Regression

- Step 1: randomly initialize the model to be trained $(a, b) = (a_0, b_0)$
- Step 2: repeat the following steps for N epochs
  - Step 2.1: randomly shuffle the dataset $\{(x_t, y_t)\}_{t=1}^{N}$
  - Step 2.2: sequentially load a mini-batch to update (a, b)

updating steps

- Step 2.2.1: compute the $L_{MSE}(a_t, b_t)$
- Step 2.2.2: compute $\frac{\partial L_{MSE}(a_t, b_t)}{\partial a}$ and $\frac{\partial L_{MSE}(a_t, b_t)}{\partial b}$
- Step 2.2.3: update (a, b)

$$a_{t+1} = a_t - \eta \times \frac{\partial L_{MSE}(a_t, b_t)}{\partial a} \qquad b_{t+1} = b_t - \eta \times \frac{\partial L_{MSE}(a_t, b_t)}{\partial b}$$

Linear Regression

- NumPy Program
  - use NumPy (Numerical Python) package to implement gradient descent algorithm

```
import numpy
```

import NumPy package

```
data_points = numpy.array([[0.11, 1.24],
                           [0.20, 1.40],
                           [0.49, 2.00],
                           [0.65, 2.27],
                           [0.67, 2.41],
                           [0.83, 2.52]])
```

initialize data_points as a 6x2 numpy array with values

x_i	0.11	0.20	0.49	0.65	0.67	0.83
y_i	1.24	1.40	2.00	2.27	2.41	2.52

initialize hyper-parameters

```
n_epoch = 4000
batch_size = 2
eta = 0.01
```

- n_epoch: number of epochs
- batch_size: number of items in a mini-batch
- eta: learning rate

```
a = numpy.random.randn(1)
b = numpy.random.randn(1)
```

Step 1: randomly initialize (a, b)
using ***randn()*** in numpy.random package

```
for epoch in range(n_epoch):
```

Step 2: repeatedly visit all items
in data_points for n_epoch times

```
    numpy.random.shuffle(data_points)
```

Step 2.1: shuffle items in data_points

```
    for start_index in range(0, len(data_points), batch_size):
        points = data_points[start_index:start_index+batch_size]
        x = points[:,0]
        y = points[:,1]
```

Step 2.2: sequentially take a mini-batch
for updating a and b

When there are many parameters to be estimated, it is hard to compute their gradients and update them, respectively

```
        a_grad = -2 * ((x * y_error).mean())
        b_grad = -2 * (y_error.mean())
```

Step 2.2.2: compute $\frac{\partial L_{MSE}(a_t,b_t)}{\partial a}$ and $\frac{\partial L_{MSE}(a_t,b_t)}{\partial b}$

```
        a = a - eta * a_grad
        b = b - eta * b_grad
```

Step 2.2.3: update (a, b)

$$a_{t+1} = a_t - \eta \times \frac{\partial L_{MSE}(a_t, b_t)}{\partial a}$$

$$a_{t+1} = a_t - \eta \times \frac{\partial L_{MSE}(a_t, b_t)}{\partial a}$$

PyTorch Programming

- Autograd and Optimizer
  - ***autograd*** package: include differential operations on PyTorch tensors
    - define the model parameters (***a*** and ***b***, in this case) as PyTorch tensors
    - convert the data for updating model parameters to PyTorch tensors
    - perform backward propagation by calling ***backward()*** of loss tensor variable (***mse_loss***, in this case)

```
a = numpy.random.randn(1)
b = numpy.random.randn(1)
```

```
a = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)
```

define model parameters (a, b)
as PyTorch tensors

```
for epoch in range(n_epoch):

    numpy.random.shuffle(data_points)

    for start_index in range(0, len(data_points), batch_size):
        points = data_points[start_index:start_index+batch_size]
        x = points[:,0]
        y = points[:,1]
```

```
        x = torch.FloatTensor(points[:,0])
        y = torch.FloatTensor(points[:,1])
```

convert the data for updating (a, b)
to PyTorch tensors

```
        y_hat = a * x + b
        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        a_grad = -2 * ((x * y_error).mean())
        b_grad = -2 * (y_error.mean())
```

```
        mse_loss.backward()

        a = a - eta * a_grad
        b = b - eta * b_grad
```

compute partial derivatives w.r.t. (a, b)
via back propagation

```
a = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

for epoch in range(n_epoch):

    numpy.random.shuffle(data_points)

    for start_index in range(0, len(data_points), batch_size):
        points = data_points[start_index:start_index+batch_size]
        x = torch.FloatTensor(points[:,0])
        y = torch.FloatTensor(points[:,1])

        y_hat = a * x + b
        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        mse_loss.backward()

        a = a - eta * a_grad
        b = b - eta * b_grad
```

PyTorch Programming

- Autograd and Optimizer
  - ***torch.optim*** package: implement various optimization algorithms, SGD, Adam, etc.
    - create an optimizer object by taking model parameters and learning rate as arguments
    - update the model parameters by invoking ***step()*** method of the optimizer
    - zero the gradients of all variables after updating by invoking ***zero_grad()*** method of the optimizer

```
a = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

for epoch in range(n_epoch):

    numpy.random.shuffle(data_points)

    for start_index in range(0, len(data_points), batch_size):
        points = data_points[start_index:start_index+batch_size]
        x = torch.FloatTensor(points[:,0])
        y = torch.FloatTensor(points[:,1])

        y_hat = a * x + b
        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        mse_loss.backward()

        a = a - eta * a_grad
        b = b - eta * b_grad
```

create an optimizer object by taking model parameters ($[a, b]$) and learning rate (eta) as arguments

```
optimizer = \
    torch.optim.SGD([a,b], eta)
```

update (a, b) by invoking ***step***() of the optimizer

```
optimizer.step()
```

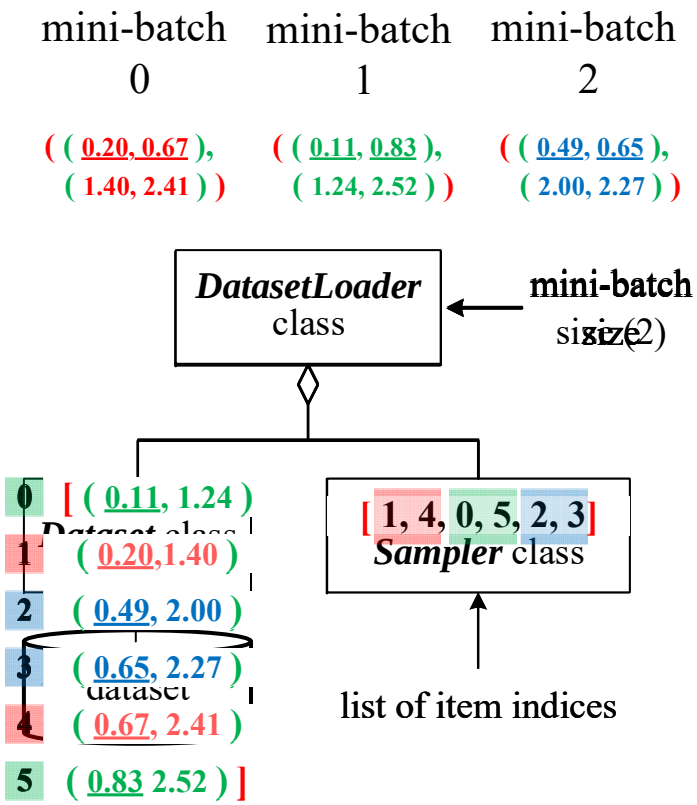
zero all gradients by invoking ***zero_grad***()

```
optimizer.zero_grad()
```

```
a = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

optimizer = \
    torch.optim.SGD([a,b], eta)

for epoch in range(n_epoch):

    numpy.random.shuffle(data_points)

    for start_index in range(0, len(data_points), batch_size):
        points = data_points[start_index:start_index+batch_size]
        x = torch.FloatTensor(points[:,0])
        y = torch.FloatTensor(points[:,1])

        y_hat = a * x + b
        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        mse_loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

PyTorch Programming

- Data Providing
  - Providing data in PyTorch are based on three classes in ***torch.utils.data*** package
    - ***Dataset*** class: have the interface to index data item(feature + label)
    - ***SubsetRandomSampler*** class: shuffle item indices to provide random sampling
    - ***DataLoader*** class: is an iterator that slices the dataset according to desired min-batch size



PyTorch Programming

- Data Providing
  - declare an class inheriting from abstract ***Dataset*** class to represent a dataset
  - implement three abstract methods
    - ***__init__(self)***: take arguments needed to build a list of tuples depending on the task at hand
    - ***__getitem__(self, index)***: return a tuple (feature, label) for the requested data at **indexth** position
    - ***__len__(self)***: return the number of items (size) in the whole dataset.

PyTorch Programming

- Data Providing
  - create an object of ***Dataset*** class
  - create an object of ***SubsetRandomSampler*** class by taking the list of item indices as argument
  - create an object of ***DataLoader*** class by taking three arguments
    - ***Dataset*** object
    - ***SubsetRandomSampler*** object
    - mini-batch size

```
class LRDataset(Dataset):
```
inherit from ***Dataset*** class

```
    def __init__(self, data_points):
        self.data_points = data_points
```
take the points for linear regression as input and store them to instance variable ***self.data_points***

define three methods

```
    def __getitem__(self, index):
        x = torch.tensor(self.data_points[index][0], dtype=torch.float)
        y = torch.tensor(self.data_points[index][1], dtype=torch.float)
        return x, y
```
return a tuple (x, y) at **index** position as PyTorch tensors

```
    def __len__(self):
        return len(self.data_points)
```
return the number of data points (in this case is 6)

```
lr_dataset = LRDataset(data_points)
```
create a ***dataset*** object as ***lr_dataset***

```
indices = list(range(len(lr_dataset)))
sampler = SubsetRandomSampler(indices)
```
generate a list of item indices and use it to create ***SubsetRandomSampler*** object as ***sampler***

```
lr_loader = DataLoader(dataset=lr_dataset, \
                       sampler=sampler,\
                       batch_size = batch_size)
```
take ***lr_dataset***, ***sampler,*** and ***batch_size*** as arguments to generate ***DataLoader*** object as ***lr_loader***

```
a = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

optimizer = \
    torch.optim.SGD([a,b], eta)

for epoch in range(n_epoch):

    numpy.random.shuffle(data_points)

    for start_index in range(0, len(data_points), batch_size):
        points = data_points[start_index:start_index+batch_size]
        x = torch.FloatTensor(points[:,0])
        y = torch.FloatTensor(points[:,1])

        y_hat = a * x + b
        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        mse_loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

➡ `for x, y in lr_loader:`

iterate all mini-batches provided by ***lr_loader***

```
a = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

optimizer = \
    torch.optim.SGD([a,b], eta)

for epoch in range(n_epoch):

    for x, y in lr_loader:

        y_hat = a * x + b

        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        mse_loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

PyTorch Programming

- Model Building
  - inherit from abstract ***Module*** class to represent a model
  - implement two abstract methods
    - ***__init__(self)***: define the model part (parameters *a* and *b* in this case)
    - ***forward(self, x)***: give the input **x** to output prediction

You should **NOT** call forward().
You should call the **whole model** itself to perform forward pass

inherit from ***Module*** class

```
class LRModel(torch.nn.Module):

    def __init__(self):
        super(LRModel, self).__init__()

    self.a = torch.nn.Parameter(torch.rand(1), \
                                requires_grad=True)
    self.b = torch.nn.Parameter(torch.rand(1), \
                                requires_grad=True)

    def forward(self, x):
        return self.a * x + self.b
```

init base class by passing class name ***LRModel*** as argument

implements two methods

randomly set variables a, b as model parameters to be estimated and turn on ***required_grad*** flag

return the current prediction

$$y = a \times x + b$$

```
lr_model = LRModel()
```

create a model as **lr_model**

```
a = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)

optimizer = \
    torch.optim.SGD([a,b], eta)

for epoch in range(n_epoch):

    for x, y in lr_loader:

        y_hat = a * x + b

        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        mse_loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

```
lr_model.parameters()
```

replace , $[a, b]$ to with the parameters in model to ***SGD*** for further optimization

```
lr_model.train()
y_hat = lr_model(x)
```

set model as train mode and predict ***y_hat*** by invoking ***forward***() method

```
optimizer = \
    torch.optim.SGD(lr_model.parameters(), eta)

for epoch in range(n_epoch):

    for x, y in lr_loader:

        lr_model.train()
        y_hat = lr_model(x)

        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        mse_loss.backward()

        optimizer.step()
        optimizer.zero_grad()
```

```
import numpy
import torch
```

import packages

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from torch.utils.data import SubsetRandomSampler
```

```
class LRDataset(Dataset):
```

declare LRDataset class

```
    def __init__(self, data_points):
        self.data_points = data_points

    def __getitem__(self, index):
        x = torch.tensor(self.data_points[index][0], \
                         dtype=torch.float)
        y = torch.tensor(self.data_points[index][1], \
                         dtype=torch.float)

        return x, y

    def __len__(self):
        return len(self.data_points)
```

```
class LRModel(torch.nn.Module):
```

declare LRModel class

```
    def __init__(self):
        super(LRModel, self).__init__()

        self.a = torch.nn.Parameter(torch.rand(1), \
                                    requires_grad=True)
        self.b = torch.nn.Parameter(torch.rand(1), \
                                    requires_grad=True)

    def forward(self, x):
        return self.a * x + self.b
```

```
n_epoch = 4000
batch_size = 2
eta = 0.01
```

set hyper-parameters

define objects for data providing

```
data_points = numpy.array([[0.11, 1.24],
                           [0.20, 1.40],[0.49, 2.00],
                           [0.65, 2.27],[0.67, 2.41],
                           [0.83, 2.52]])
```

```
lr_dataset = LRDataset(data_points)

index = list(range(len(lr_dataset)))
sampler = SubsetRandomSampler(index)
lr_loader = DataLoader(dataset=lr_dataset, \
                       batch_size = batch_size, \
                       sampler=sampler)
```

```
lr_model = LRModel()
```

define model and optimizer

```
optimizer = \
    torch.optim.SGD(lr_model.parameters(), eta)
```

```
for epoch in range(n_epoch):

    for x, y in lr_loader:

        lr_model.train()
        y_hat = lr_model(x)

        y_error = y - y_hat
        mse_loss = (y_error **2).mean()

        mse_loss.backward()

        optimizer.step()
        optimizer.zero_grad()

    print(f'\n{epoch}:{lr_model.a}:{lr_model.b}')
```

run learning paradigm


thank you
© 1996, 2002 SANRIO CO., LTD.