



PyTorch Tutorial

05. Linear Regression with PyTorch

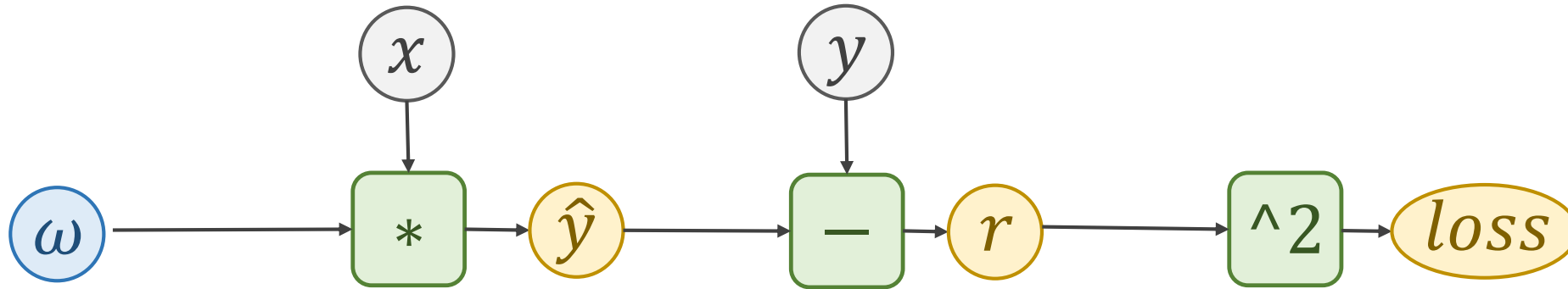
Revision

Linear Model

$$\hat{y} = x * \omega$$

Loss Function

$$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$$



Revision

```
print("predict (before training)", 4, forward(4).item())

for epoch in range(100):
    for x, y in zip(x_data, y_data):
        l = loss(x, y)      Forward
        l.backward()        Backward
        print('\tgrad:', x, y, w.grad.item())
        w.data = w.data - 0.01 * w.grad.data  update

        w.grad.data.zero_() 手动清零

    print("progress:", epoch, l.item())

print("predict (after training)", 4, forward(4).item())
```

```
predict (before training) 4 4.0
      grad: 1.0 2.0 -2.0
      grad: 2.0 4.0 -7.840000152587891
      grad: 3.0 6.0 -16.228801727294922
progress: 0 7.315943717956543
      grad: 1.0 2.0 -1.478623867034912
      grad: 2.0 4.0 -5.796205520629883
      grad: 3.0 6.0 -11.998146057128906
progress: 1 3.9987640380859375
      grad: 1.0 2.0 -1.0931644439697266
      grad: 2.0 4.0 -4.285204887390137
      grad: 3.0 6.0 -8.870372772216797
progress: 2 2.1856532096862793
      grad: 1.0 2.0 -0.8081896305084229
      grad: 2.0 4.0 -3.1681032180786133
      grad: 3.0 6.0 -6.557973861694336
progress: 3 1.1946394443511963
      grad: 1.0 2.0 -0.5975041389465332
      grad: 2.0 4.0 -2.3422164916992188
      grad: 3.0 6.0 -4.848389625549316
progress: 4 0.6529689431190491
      grad: 1.0 2.0 -0.4417421817779541
      grad: 2.0 4.0 -1.7316293716430664
      grad: 3.0 6.0 -3.58447265625
progress: 5 0.35690122842788696
      grad: 1.0 2.0 -0.3265852928161621
      grad: 2.0 4.0 -1.2802143096923828
      grad: 3.0 6.0 -2.650045394897461
```

1

Prepare dataset
we shall talk about this later

2

Design model using Class
inherit from nn.Module 计算 \hat{y}

3

Construct loss and optimizer
using PyTorch API

4

Training cycle
forward, backward, update

PyTorch直接写`**`表示举证对应位置上元素相乘，矩阵乘法对应于`torch.matmul`.

Linear Regression – 1. Prepare dataset

In PyTorch, the computational graph is in **mini-batch** fashion, so X and Y are 3×1 Tensors.

$$\begin{bmatrix} y_{pred}^{(1)} \\ y_{pred}^{(2)} \\ y_{pred}^{(3)} \end{bmatrix} = \omega \cdot \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \end{bmatrix} + b$$

```
import torch

x_data = torch.Tensor([[1.0], [2.0], [3.0]])
y_data = torch.Tensor([[2.0], [4.0], [6.0]])
```

Revision: Gradient Descent Algorithm

Derivative

$$\begin{aligned}\frac{\partial cost(\omega)}{\partial \omega} &= \frac{\partial}{\partial \omega} \frac{1}{N} \sum_{n=1}^N (x_n \cdot \omega - y_n)^2 \\ &= \frac{1}{N} \sum_{n=1}^N \frac{\partial}{\partial \omega} (x_n \cdot \omega - y_n)^2 \\ &= \frac{1}{N} \sum_{n=1}^N 2 \cdot (x_n \cdot \omega - y_n) \frac{\partial (x_n \cdot \omega - y_n)}{\partial \omega} \\ &= \frac{1}{N} \sum_{n=1}^N 2 \cdot x_n \cdot (x_n \cdot \omega - y_n)\end{aligned}$$

Gradient

$$\frac{\partial cost}{\partial \omega}$$

Update

$$\omega = \omega - \alpha \frac{\partial cost}{\partial \omega}$$

Update

$$\omega = \omega - \alpha \frac{1}{N} \sum_{n=1}^N 2 \cdot x_n \cdot (x_n \cdot \omega - y_n)$$

Linear Regression – 2. Design Model

Affine Model = Linear Unit

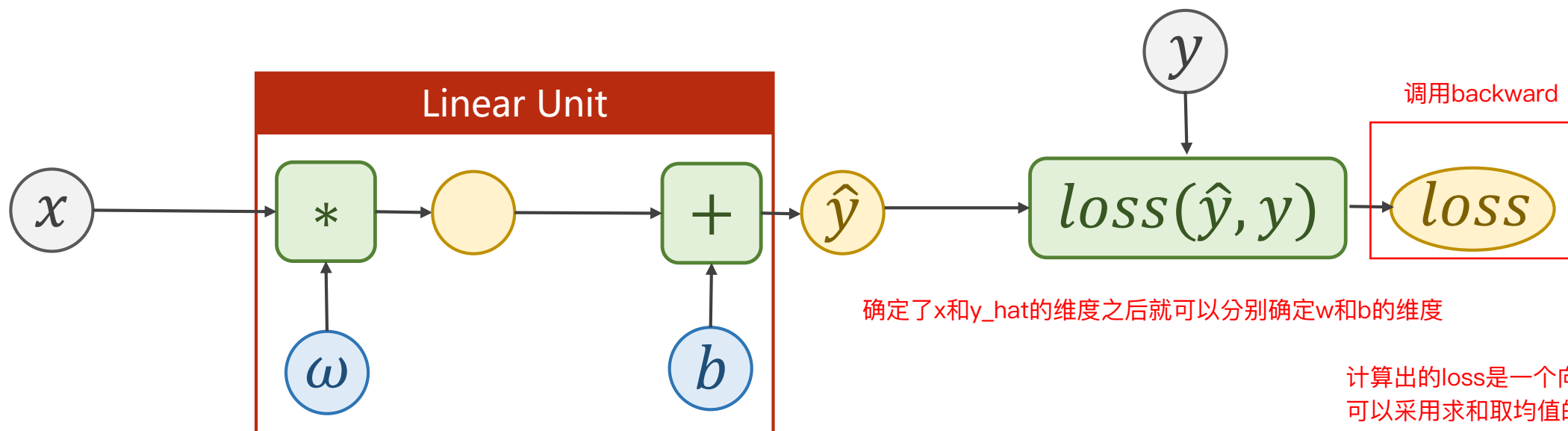
Affine Model

$$\hat{y} = x * \omega + b$$

权重 偏置

Loss Function

$$loss = (\hat{y} - y)^2 = (x \cdot \omega - y)^2$$



Linear Regression – 2. Design Model

PyTorch中自定义的所有模型类都需要继承自nn.Module

将模型定义为一个类

```
class LinearModel(torch.nn.Module):  
    def __init__(self): 构造函数  
        super(LinearModel, self).__init__()  
        self.linear = torch.nn.Linear(1, 1)  
  
    def forward(self, x):  
        y_pred = self.linear(x)  
        return y_pred  
  
model = LinearModel()
```

Our model class should be inherit from *nn.Module*, which is Base class for all neural network modules.


至少需要__init__和forward两个函数

Linear Regression – 2. Design Model

```
class LinearModel(torch.nn.Module):  
    def __init__(self):  
        super(LinearModel, self).__init__()  
        self.linear = torch.nn.Linear(1, 1)  
  
    def forward(self, x):  
        y_pred = self.linear(x)  
        return y_pred  
  
model = LinearModel()
```

Member methods *__init__()* and *forward()* have to be implemented.

Linear Regression – 2. Design Model

```
class LinearModel(torch.nn.Module):  
    def __init__(self):  
        super(LinearModel, self).__init__()   
        self.linear = torch.nn.Linear(1, 1)  
        构造对象，传入in_feature和out_feature  
  
    def forward(self, x):  
        y_pred = self.linear(x)  
        return y_pred  
  
model = LinearModel()
```

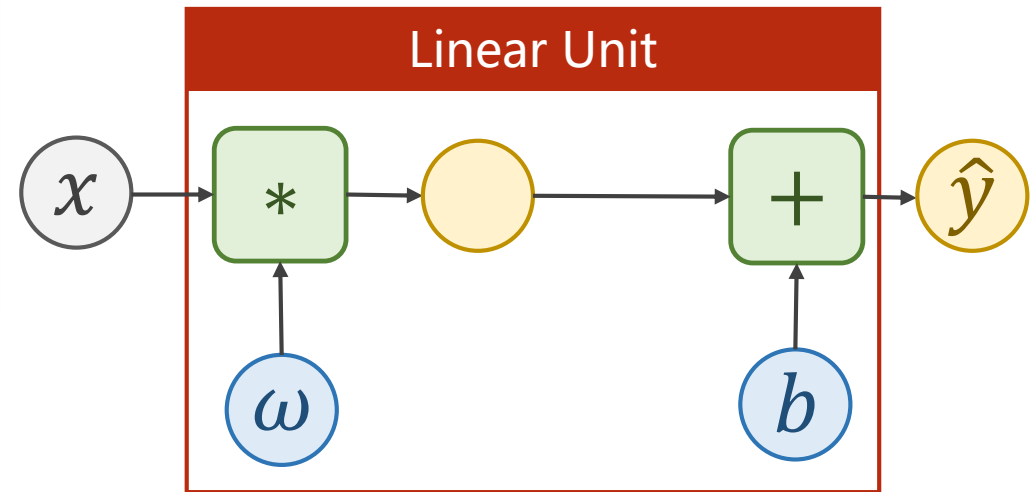
Just do it. :)

in_feature: x是几维的
out_feature: y是几维的

Linear Regression – 2. Design Model

```
class LinearModel(torch.nn.Module):  
    def __init__(self):  
        super(LinearModel, self).__init__()  
        self.linear = torch.nn.Linear(1, 1)  
  
    def forward(self, x):  
        y_pred = self.linear(x)  
        return y_pred  
  
model = LinearModel()
```

Class *nn.Linear* contain two member **Tensors**: **weight** and **bias**.



Linear Regression – 2. Design Model

```
class torch.nn.Linear(in_features, out_features, bias=True) \[source\]
```

Applies a linear transformation to the incoming data: $y = Ax + b$

Parameters:

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to False, the layer will not learn an additive bias. Default: **True**

是否需要偏置量

Shape:

- Input: $(N, *, in_features)$ where $*$ means any number of additional dimensions
- Output: $(N, *, out_features)$ where all but the last dimension are the same shape as the input.

Variables:

- **weight** – the learnable weights of the module of shape $(out_features \times in_features)$
- **bias** – the learnable bias of the module of shape $(out_features)$

Linear Regression – 2. Design Model

```
class torch.nn.Linear(in_features, out_features, bias=True) \[source\]
```

Applies a linear transformation to the incoming data: $y = Ax + b$

Parameters:

$$\begin{bmatrix} y_{pred}^{(1)} \\ y_{pred}^{(2)} \\ y_{pred}^{(3)} \end{bmatrix} = \omega \cdot \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \end{bmatrix} + b$$

The equation is presented within a yellow rectangular box. A red box labeled "Output" is positioned to the left of the output vector, and a red box labeled "Input" is positioned to the right of the input vector.

...tive bias. Default: **True**

Shape:

- Input: $(N, *, in_features)$ where $*$ means any number of additional dimensions
- Output: $(N, *, out_features)$ where all but the last dimension are the same shape as the input.

Variables:

- **weight** – the learnable weights of the module of shape $(out_features \times in_features)$
- **bias** – the learnable bias of the module of shape $(out_features)$

Linear Regression – 2. Design Model

```
class LinearModel(torch.nn.Module):  
    def __init__(self):  
        super(LinearModel, self).__init__()  
        self.linear = torch.nn.Linear(1, 1)  
  
    def forward(self, x): self.linear是一个具体对象  
        y_pred = self.linear(x) ← 对象后面加括号意味着实现了一个callable的对象，可以让对象像函数一样被调用  
        return y_pred  
  
model = LinearModel()
```

Class *nn.Linear* has implemented the magic method *__call__()*, which enable the instance of the class can be called just like a function. Normally the *forward()* will be called.

Pythonic!!!

Linear Regression – 2. Design Model

```
class LinearModel(torch.nn.Module):  
    def __init__(self):  
        super(LinearModel, self).__init__()  
        self.linear = torch.nn.Linear(1, 1)  
  
    def forward(self, x):  
        y_pred = self.linear(x)  
        return y_pred
```


```
model = LinearModel()
```

Create a instance of class
LinearModel.

Linear Regression – 3. Construct Loss and Optimizer

构造损失函数和优化器

```
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```



```
class torch.nn.MSELoss(size_average=True, reduce=True) \[source\]
```

Creates a criterion that measures the mean squared error between target y .

The loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size.

Also inherit from **nn.Module**.

Linear Regression – 3. Construct Loss and Optimizer

```
criterion = torch.nn.MSELoss(size_average=False)

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```


lr=学习率

`class torch.optim.SGD(params, lr=<object object>, momentum=0, dampening=0, weight_decay=0, nesterov=False)` [\[source\]](#)

Implements stochastic gradient descent (optionally with momentum).

Linear Regression – 3. Construct Loss and Optimizer

```
criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```



Parameters:

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float*) – learning rate 可以设置学习率为固定，也可以将学习率设置为先高后低

Linear Regression – 4. Training Cycle

```
for epoch in range(100):  
    y_pred = model(x_data)  
    loss = criterion(y_pred, y_data)  
    print(epoch, loss)  
  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()
```

Forward: Predict



Linear Regression – 4. Training Cycle

```
for epoch in range(100):  
    y_pred = model(x_data)  
    loss = criterion(y_pred, y_data)  
    print(epoch, loss)
```

```
optimizer.zero_grad() 所有权重归零  
loss.backward()  
optimizer.step()
```

Loss虽然是一个tensor, 但是在打印时会自动转换为str

Forward: Loss

Linear Regression – 4. Training Cycle

```
for epoch in range(100):  
    y_pred = model(x_data)  
    loss = criterion(y_pred, y_data)  
    print(epoch, loss)  
  
    optimizer.zero_grad() ←  
    loss.backward()  
    optimizer.step()
```

NOTICE:

The grad computed by *.backward()* will be **accumulated**.

So before backward, remember set the grad to **ZERO!!!**

Linear Regression – 4. Training Cycle

```
for epoch in range(100):  
    y_pred = model(x_data)  
    loss = criterion(y_pred, y_data)  
    print(epoch, loss)  
  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()
```

Backward: Autograd



Linear Regression – 4. Training Cycle

```
for epoch in range(100):  
    y_pred = model(x_data)  
    loss = criterion(y_pred, y_data)  
    print(epoch, loss)
```

```
optimizer.zero_grad()  
loss.backward()  
optimizer.step()
```

```
for x, y in zip(x_data, y_data):  
    .....  
    w.data = w.data - 0.01 * w.grad.data
```

Update



Linear Regression – Test Model

```
# Output weight and bias
print('w = ', model.linear.weight.item())
print('b = ', model.linear.bias.item())

# Test Model
x_test = torch.Tensor([[4.0]])
y_test = model(x_test)
print('y_pred = ', y_test.data)
```

```
86 0.3036523759365082
87 0.2992883026599884
88 0.29498720169067383
89 0.2907477021217346
90 0.28656935691833496
91 0.28245046734809875
92 0.27839142084121704
93 0.27439042925834656
94 0.2704470157623291
95 0.2665606141090393
96 0.262729674577713
97 0.25895369052886963
98 0.2552322745323181
99 0.2515641450881958
w = 1.666100263595581
b = 0.7590328454971313
y_pred = tensor([[ 7.4234]])
```

100 Iterations

```
986 3.594939812501252e-07
987 3.5411068211033125e-07
988 3.4917979974125046e-07
989 3.4428359185767476e-07
990 3.392528924450744e-07
991 3.3442694302721065e-07
992 3.294019847999152e-07
993 3.247135396122758e-07
994 3.199925231456291e-07
995 3.1540417921860353e-07
996 3.1097857799977646e-07
997 3.0668098816022393e-07
998 3.020934400410624e-07
999 2.977626536448952e-07
w = 1.9996366500854492
b = 0.0008257834706455469
y_pred = tensor([[ 7.9994]])
```

1000 Iterations

Linear Regression

```
import torch

x_data = torch.Tensor([[1.0], [2.0], [3.0]])
y_data = torch.Tensor([[2.0], [4.0], [6.0]])

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

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

model = LinearModel()

criterion = torch.nn.MSELoss(size_average=False)
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for epoch in range(1000):
    y_pred = model(x_data)
    loss = criterion(y_pred, y_data)
    print(epoch, loss.item())

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

print('w = ', model.linear.weight.item())
print('b = ', model.linear.bias.item())

x_test = torch.Tensor([[4.0]])
y_test = model(x_test)
print('y_pred = ', y_test.data)
```

1

Prepare dataset
we shall talk about this later

2

Design model using Class
inherit from nn.Module

3

Construct loss and optimizer
using PyTorch API

4

Training cycle
forward, backward, update

Exercise 5-1: Try Different Optimizer in Linear Regression

- `torch.optim.Adagrad`
- `torch.optim.Adam`
- `torch.optim.Adamax`
- `torch.optim.ASGD`
- `torch.optim.LBFGS`
- `torch.optim.RMSprop`
- `torch.optim.Rprop`
- `torch.optim.SGD`

Exercise 5-2: Read more example from official tutorial

Table of Contents

- Tensors
 - Warm-up: numpy
 - PyTorch: Tensors
- Autograd
 - PyTorch: Tensors and autograd
 - PyTorch: Defining new autograd functions
 - TensorFlow: Static Graphs
- *nn* module
 - PyTorch: nn
 - PyTorch: optim
 - PyTorch: Custom nn Modules
 - PyTorch: Control Flow + Weight Sharing
- Examples
 - Tensors
 - Autograd
 - *nn* module

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html



PyTorch Tutorial

05. Linear Regression with PyTorch