

torch.optim.SGD()

Note that the SGD in Pytorch means the mini-batch gradient descent.

The Basic Process in torch.optim.SGD()

Let's consider a regression problem, assuming that our data includes 6 samples with 1 dimension:

$$\{(x, y), x \in R^1, y \in R^1 | (1, 2), (2, 2.8), (3, 3.6), (4, 4.4), (5, 5.2)\} \quad (1)$$

We can see the equation between x and y is $y = 0.8 \times x + 1.2$. Now, Let's set a **linear regression** model, the formula can be expressed as:

$$\hat{y} = w \cdot x + b \quad (2)$$

Set the **MSE Loss** as the objective function, the formula can be expressed as: (since the dim of data is 1, so we can re-formula it as below)

$$\begin{aligned} \mathcal{L}(w, b) &= MSE(\hat{y}, y) = \sum_{i=0}^d (\hat{y}_i - y_i)^2 \\ &= (\hat{y} - y)^2 \\ &= (w \cdot x + b - y)^2 \end{aligned} \quad (3)$$

And we can calculate the gradient of w and b in this example:

$$\begin{cases} \nabla_{w_t} \mathcal{L} = 2 \cdot (w \cdot x + b - y) \cdot x \\ \nabla_{b_t} \mathcal{L} = 2 \cdot (w \cdot x + b - y) \end{cases} \quad (4)$$

Let's use **Stochastic Gradient Descent** algorithm to solve the parameters of model (η is the learning rate), which will choose only one sample to calculate gradient:

$$\begin{cases} w_{t+1} = w_t - \eta \cdot \nabla_{w_t} \mathcal{L} \\ b_{t+1} = b_t - \eta \cdot \nabla_{b_t} \mathcal{L} \end{cases} \quad (5)$$

In this example, the iterative formula is:

$$\begin{cases} w_{t+1} = w_t - \eta \cdot 2(w_t \cdot x + b_t - y) \cdot x \\ b_{t+1} = b_t - \eta \cdot 2(w_t \cdot x + b_t - y) \end{cases} \quad (6)$$

Let's use python to simulate the process.

Firstly, the definition of the Dataset:

```

1  from torch.utils.data import Dataset
2
3  class MyData(Dataset):
4      def __init__(self):
5          super(MyData, self).__init__()
6          self.data = torch.tensor([[1], [2], [3], [4], [5]],
dtype=torch.float)
7          self.label = 0.8 * self.data + 1.2
8
9      def __getitem__(self, item):
10         return self.data[item], self.label[item]
11
12     def __len__(self):
13         return len(self.data)

```

Then, the definition of the Linear Regression Model:

```

1  class MyModel(nn.Module):
2      def __init__(self):
3          super(MyModel, self).__init__()
4          self.layer = nn.Linear(1, 1)
5
6      def forward(self, x):
7          return self.layer(x)

```

the main function as below:

```

1  # ---A Simple Framework of the plain Gradient Descent ---#
2  model = MyModel()
3  data = MyData()
4  # learning rate
5  LR = 0.1
6
7  optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=0,
weight_decay=0)
8  # the w,b for iteration
9  w = model.layer.weight.data
10 b = model.layer.bias.data
11 for _ in range(5):
12     print("====={}th data=====".format(_))
13     # the process of official Pytorch
14     inputs, target = data[_]
15     output = model(inputs)
16     loss = F.mse_loss(output, target)
17     print("Pytorch mseloss:", loss)
18     print("Our mseloss:", (w * inputs + b - target) ** 2)
19     optimizer.zero_grad()
20     loss.backward()
21     optimizer.step()
22
23     # the grad
24     print("Pytorch w_grad:", model.layer.weight.grad)
25     print("Cal w_grad:", 2 * (w * inputs + b - target) * inputs)
26     print("Pytorch b_grad:", model.layer.bias.grad)
27     print("Cal b_grad:", 2 * (w * inputs + b - target)
28

```

```

29
30 # -----our iterative formula-----#
31 new_w = w - LR * 2 * (w * inputs + b - target) * inputs
32 new_b = b - LR * 2 * (w * inputs + b - target)
33 print("Pytorch new_w:", model.layer.weight.data)
34 print("Our new_w:", new_w)
35 print("Pytorch new_b:", model.layer.bias.data)
36 print("Our new_b:", new_b)
37 w = new_w
38 b = new_b
39 print(model.layer.weight.data, model.layer.bias.data)
40 print(w, b)

```

We can get `tensor[[0.9559]]` `tensor[1.4754]` and `tensor[0.9559]` `tensor[1.4754]`. Our result is the same as the Pytorch official implement. The main iterative formula is : (corresponding the formula (6).)

```

1 new_w = w - LR * 2 * (w * inputs.item() + b - target.item()) * inputs.item()
2 new_b = b - LR * 2 * (w * inputs.item() + b - target.item())

```

The weight_decay in torch.optim.SGD()

The same situation as above, but we set the weight decay in SGD to be 0.5 rather than 0.

Consider the L2 penalty in MSE Loss:

$$\mathcal{L}'(w, b) = MSE'(\hat{y}, y) = \sum_{i=0}^d (\hat{y}_i - y_i)^2 + \frac{\lambda}{2d} \cdot ||w||^2 \quad (7)$$

$$= (\hat{y} - y)^2 + \frac{\lambda}{2d} \cdot ||w||^2 \quad (8)$$

$$= (w \cdot x + b - y)^2 + \frac{\lambda}{2d} \cdot ||w||^2 \quad (9)$$

$$= (w \cdot x + b - y)^2 + \frac{\lambda}{2} \cdot ||w||^2 \quad (10)$$

$$= \mathcal{L}(w, b) + \frac{\lambda}{2} \cdot w^2 \quad (11)$$

And iterative formula will be: (the λ is the weight_decay in SGD)

$$\begin{aligned}
w_{t+1} &= w_t - \eta \cdot \nabla_{w_t} \mathcal{L}' \\
&= w_t - \eta \cdot (\nabla_{w_t} \mathcal{L} + \lambda \cdot w) \\
&= w_t - \eta \cdot \nabla_{w_t} \mathcal{L} - \eta \cdot \lambda \cdot w \\
&= (1 - \eta\lambda) \cdot w_t - \eta \cdot \nabla_{w_t} \mathcal{L}
\end{aligned} \quad (12)$$

$$\text{Main Formulation: } w_{t+1} = w_t - \eta \cdot (\nabla_{w_t} \mathcal{L} + \lambda \cdot w)$$

Note that the weight decay in SGD is equivalent to the L2 penalty.

So we re-formula our iterative formulation in our code:

```

1 WD = 0.5
2 optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=0,
  weight_decay=WD)
3 ...
4 new_w = (1 - LR * WD) * w - LR * 2 * (w * inputs.item() + b - target.item())
  * inputs.item()
5 new_b = (1 - LR * WD) * b - LR * 2 * (w * inputs.item() + b - target.item())
6 ...
7 w = new_w
8 b = new_b

```

We can get `tensor[[0.6061]]` `tensor[0.9894]` and `tensor[0.6061]` `tensor[0.9894]`. Our result is the same as the Pytorch official implement.

The Momentum in torch.optim.SGD()

The same situation as above, but we set the momentum in SGD to be 0.9 rather than 0.

Consider the Exponential moving average of the gradient: (β is the momentum)

$$W_0 = \nabla_{w_0} \mathcal{L} \quad (13)$$

$$W_t = \beta \cdot W_{t-1} + (1 - \beta) \cdot \nabla_{w_t} \mathcal{L} \quad (14)$$

And our iterative formulation is:

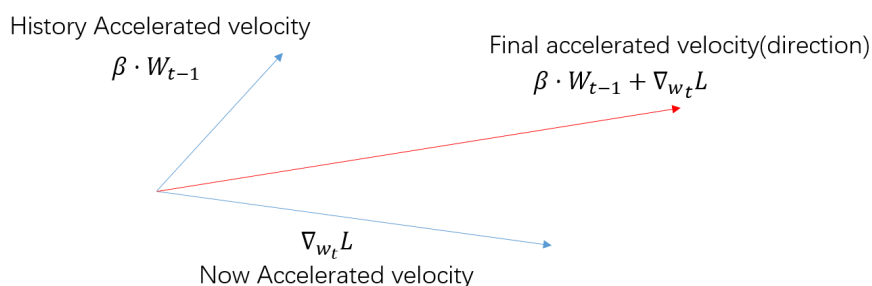
$$\begin{aligned} w_{t+1} &= w_t - \eta \cdot W_t \\ &= w_t - \eta \cdot (\beta \cdot \nabla_{w_{t-1}} \mathcal{L} + (1 - \beta) \cdot \nabla_{w_t} \mathcal{L}) \end{aligned} \quad (15)$$

BUT in Pytorch, the momentum of SGD is not same as the form of EMA. The **official Pytorch implement** lies as below (β is the momentum, d is the *dampening* parameter in `torch.optim.SGD`). The *dampening* parameter is a parameter that describes the damping of the gradient of parameter of the current time step, which is usually set to 0. In the rest of this blog, we default the d is 0.

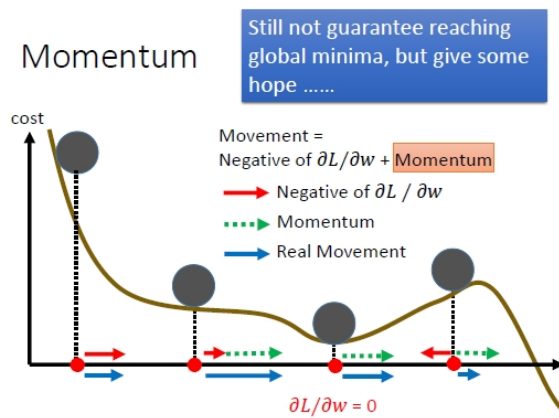
$$\begin{aligned} W_0 &= \nabla_{w_0} \mathcal{L} \\ W_t &= \beta \cdot W_{t-1} + (1 - d) \cdot \nabla_{w_t} \mathcal{L} \\ W_t &= \beta \cdot W_{t-1} + \nabla_{w_t} \mathcal{L} \end{aligned} \quad (16)$$

$$\begin{aligned} w_{t+1} &= w_t - \eta \cdot W_t \\ &= w_t - \eta \cdot (\beta \cdot W_{t-1} + (1 - d) \cdot \nabla_{w_t} \mathcal{L}) \\ &= w_t - \eta \cdot (\beta \cdot W_{t-1} + \nabla_{w_t} \mathcal{L}) \end{aligned} \quad (17)$$

We can assume that the gradient of the parameters of current time step is the accelerated velocity of the model, which indicates the direction of the next move of the model. And the momentum of model is the history trajectory of the model. The basic idea of momentum is that if a ball slides down from a slope, its next move not only related to the accelerated velocity at this time, and its history velocity.



It can help model skip out of the local minimum.



So we re-formula our iterative formulation in our code:

```
1 M = 0.9
2 optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=M,
3 weight_decay=0)
4 w_last = 0.
5 B_last = 0.
6 ...
7 w = M * w_last + 2 * (w * inputs + b - target) * inputs
8 B = M * B_last + 2 * (w * inputs + b - target)
9 new_w = w - LR * w
10 new_b = b - LR * B
11 w_last = w
12 B_last = B
13 w = new_w
14 b = new_b
```

We can get `tensor[[5.5695]] tensor[3.4704]` and `tensor[5.5695] tensor[3.4704]`. Our result is the same as the Pytorch official implement.

The Final form of SGD(including momentum,weight_decay)

```
1 import torch
2 from torch.utils.data import Dataset
3 import torch.nn as nn
4 import torch.nn.functional as F
5
6
7 class MyModel(nn.Module):
8     def __init__(self):
9         super(MyModel, self).__init__()
10        self.layer = nn.Linear(1, 1)
11
12    def forward(self, x):
13        return self.layer(x)
14
15
16 class MyData(Dataset):
17     def __init__(self):
18         super(MyData, self).__init__()
```

```

19         self.data = torch.tensor([[1], [2], [3], [4], [5]],
dtype=torch.float)
20         self.label = 0.8 * self.data + 1.2
21
22     def __getitem__(self, item):
23         return self.data[item], self.label[item]
24
25     def __len__(self):
26         return len(self.data)
27
28
29 model = MyModel()
30 data = MyData()
31 # learning rate
32 LR = 0.1
33 # weight decay
34 WD = 0.5
35 # momentum
36 M = 0.9
37
38 optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=M,
weight_decay=WD, dampening=0, nesterov=False)
39 w = model.layer.weight.data.item()
40 b = model.layer.bias.data.item()
41
42 # Official Pytorch Implement
43 for _ in range(5):
44     inputs, target = data[_]
45     output = model(inputs)
46     loss = F.mse_loss(output, target)
47
48     optimizer.zero_grad()
49     loss.backward()
50     optimizer.step()
51 print(model.layer.weight.data, model.layer.bias.data)
52
53 # Our Code
54 w_grad_last = 0.
55 b_grad_last = 0.
56 for _ in range(5):
57     x, y = data[_]
58     w_grad = 2 * (w * x + b - y) * x
59     b_grad = 2 * (w * x + b - y)
60     if WD != 0:
61         w_grad = w_grad + WD * w
62         b_grad = b_grad + WD * b
63     if M != 0:
64         w_grad = M * w_grad_last + w_grad
65         b_grad = M * b_grad_last + b_grad
66     new_w = w - LR * w_grad
67     new_b = b - LR * b_grad
68
69     w = new_w
70     b = new_b
71     w_grad_last = w_grad
72     b_grad_last = b_grad
73 print(w, b)

```

There is a parameter in torch.optim.SGD named "Nesterov" we don't referred.

Ref

- [基于Pytorch源码对SGD、momentum、Nesterov学习](#)
- [怎么理解Pytorch中对Nesterov的实现?](#)
- [深度学习中优化方法——momentum、Nesterov Momentum、AdaGrad、Adadelata、RMSprop、Adam](#)
- [An overview of gradient descent optimization algorithms](#)
- Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $O(1/k^2)$. Doklady ANSSSR (translated as Soviet.Math.Docl.), 269:543-547.
- (The source code of SGD.)

```
def sgd(params: List[Tensor],
        d_p_list: List[Tensor],
        momentum_buffer_list: List[Optional[Tensor]],
        *,
        weight_decay: float,
        momentum: float,
        lr: float,
        dampening: float,
        nesterov: bool):
    r"""Functional API that performs SGD algorithm computation.

    See :class:`~torch.optim.SGD` for details.
    """

    for i, param in enumerate(params):
        d_p = d_p_list[i]
        if weight_decay != 0:
            d_p = d_p.add(param, alpha=weight_decay)

        if momentum != 0:
            buf = momentum_buffer_list[i]

            if buf is None:
                buf = torch.clone(d_p).detach()
                momentum_buffer_list[i] = buf
            else:
                buf.mul_(momentum).add_(d_p, alpha=1 - dampening)

            if nesterov:
                d_p = d_p.add(buf, alpha=momentum)
            else:
                d_p = buf

        param.add_(d_p, alpha=-lr)
```