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 out data includes 6 samples with 1 dimension:

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

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:

$$\hat{y} = w \cdot x + b \tag{2}$$

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)

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

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

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

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:

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

In this example, the iterative formula is:

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

Let's use python to simulate the process.

Firstly, the definition of the Dataset:

```python
from torch.utils.data import Dataset

class MyData(Dataset):
    def __init__(self):
        super(MyData, self).__init__()
        self.data = torch.tensor([[1], [2], [3], [4], [5]], dtype=torch.float)
        self.label = 0.8 * self.data + 1.2

    def __getitem__(self, item):
        return self.data[item], self.label[item]

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

Then, the definition of the Linear Regression Model:

```python
class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.layer = nn.Linear(1, 1)

    def forward(self, x):
        return self.layer(x)
```

the main function as below:

```python
# ---A Simple Framework of the plain Gradient Descent ---#
model = MyModel()
data = MyData()
# learning rate
LR = 0.1

optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=0, weight_decay=0)
# the w,b for iteration
w = model.layer.weight.data
b = model.layer.bias.data
for _ in range(5):
    print("==========={}th data========".format(_))
    # the process of official Pytorch
    inputs, target = data[_]
    output = model(inputs)
    loss = F.mse_loss(output, target)
    print("Pytorch mseloss:", loss)
    print("Our mseloss:", (w * inputs + b - target) ** 2)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # the grad
    print("Pytorch w_grad:", model.layer.weight.grad)
    print("Cal w_grad:", 2 * (w * inputs + b - target) * inputs)
    print("Pytorch b_grad:", model.layer.bias.grad)
    print("Cal b_grad:", 2 * (w * inputs + b - target)

```

```python

    # -----our iterative formula-----#
    new_w = w - LR * 2 * (w * inputs + b - target) * inputs
    new_b = b - LR * 2 * (w * inputs + b - target)
    print("Pytorch new_w:", model.layer.weight.data)
    print("Our new_w:", new_w)
    print("Pytorch new_b:", model.layer.bias.data)
    print("Our new_b:", new_b)
    w = new_w
    b = new_b
print(model.layer.weight.data, model.layer.bias.data)
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).)

```python
new_w = w - LR * 2 * (w * inputs.item() + b - target.item()) * inputs.item()
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:

$$
\begin{aligned}
\mathcal{L}'(w,b) = MSE'(\hat{y},y) &= \sum_{i=0}^{d}(\hat{y}_i - y_i)^2 + \frac{\lambda}{2d}\cdot||w||^2 && (7)\\
&= (\hat{y}-y)^2 + \frac{\lambda}{2d}\cdot||w||^2 && (8)\\
&= (w\cdot x + b - y)^2 + \frac{\lambda}{2d}\cdot||w||^2 && (9)\\
&= (w\cdot x + b - y)^2 + \frac{\lambda}{2}\cdot||w||^2 && (10)\\
&= \mathcal{L}(w,b) + \frac{\lambda}{2}\cdot w^2 && (11)
\end{aligned}
$$

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

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

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

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

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

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

$$W_0 = \nabla_{w_0}\mathcal{L} \tag{13}$$
$$W_t = \beta \cdot W_{t-1} + (1-\beta)\cdot \nabla_{w_t}\mathcal{L} \tag{14}$$

And our iterative formulation is:

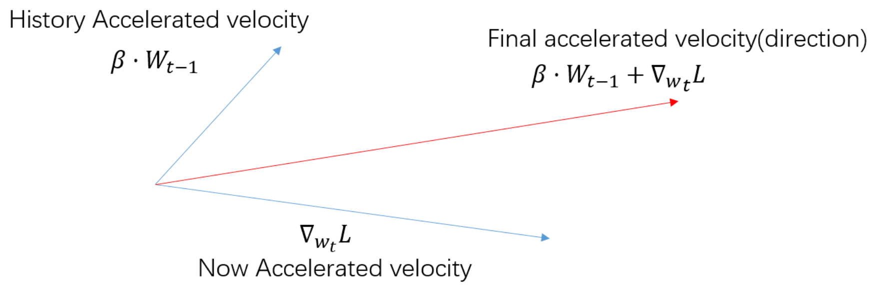
$$\begin{aligned} w_{t+1} &= w_t - \eta \cdot W_t \\ &= w_t - \eta \cdot (\beta \cdot \nabla_{w_{t-1}}\mathcal{L} + (1-\beta)\cdot \nabla_{w_t}\mathcal{L}) \end{aligned} \tag{15}$$

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.

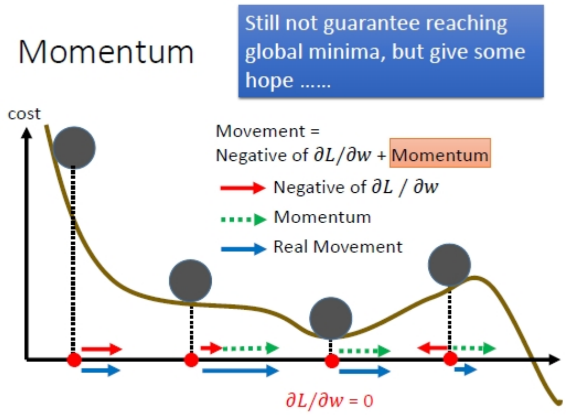
$$\begin{aligned} W_0 &= \nabla_{w_0}\mathcal{L} \\ W_t &= \beta \cdot W_{t-1} + (1-d)\cdot \nabla_{w_t}\mathcal{L} \\ W_t &= \beta \cdot W_{t-1} + \nabla_{w_t}\mathcal{L} \end{aligned} \tag{16}$$

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

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, it 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:

```
M = 0.9
optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=M, 
weight_decay=0)
W_last = 0.
B_last = 0.
...
W = M * W_last + 2 * (w * inputs + b - target) * inputs
B = M * B_last + 2 * (w * inputs + b - target)
new_w = w - LR * W
new_b = b - LR * B

W_last = W
B_last = B
w = new_w
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)

```
import torch
from torch.utils.data import Dataset
import torch.nn as nn
import torch.nn.functional as F


class MyModel(nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.layer = nn.Linear(1, 1)

    def forward(self, x):
        return self.layer(x)


class MyData(Dataset):
    def __init__(self):
        super(MyData, self).__init__()
```

```python
        self.data = torch.tensor([[1], [2], [3], [4], [5]], dtype=torch.float)
        self.label = 0.8 * self.data + 1.2

    def __getitem__(self, item):
        return self.data[item], self.label[item]

    def __len__(self):
        return len(self.data)


model = MyModel()
data = MyData()
# learning rate
LR = 0.1
# weight decay
WD = 0.5
# momentum
M = 0.9

optimizer = torch.optim.SGD(model.parameters(), lr=LR, momentum=M, weight_decay=WD, dampening=0, nesterov=False)
w = model.layer.weight.data.item()
b = model.layer.bias.data.item()

# Official Pytorch Implement
for _ in range(5):
    inputs, target = data[_]
    output = model(inputs)
    loss = F.mse_loss(output, target)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
print(model.layer.weight.data, model.layer.bias.data)

# Our Code
w_grad_last = 0.
b_grad_last = 0.
for _ in range(5):
    x, y = data[_]
    w_grad = 2 * (w * x + b - y) * x
    b_grad = 2 * (w * x + b - y)
    if WD != 0:
        w_grad = w_grad + WD * w
        b_grad = b_grad + WD * b
    if M != 0:
        w_grad = M * w_grad_last + w_grad
        b_grad = M * b_grad_last + b_grad
    new_w = w - LR * w_grad
    new_b = b - LR * b_grad

    w = new_w
    b = new_b
    w_grad_last = w_grad
    b_grad_last = b_grad
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、Adadelta、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/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), 269:543–547.

- (The source code of SGD.)

```python
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)
```