In [1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

In [2]:
```
x_train = torch.FloatTensor([[1], [2], [3]])
y_train = torch.FloatTensor([[2], [4], [6]])
```

In [3]:
```
x_train
```

Out[3]:
```
tensor([[1.],
        [2.],
        [3.]])
```

In [4]:
```
W = torch.zeros(1, requires_grad = True)
W
```

Out[4]:
```
tensor([0.], requires_grad=True)
```

In [5]:
```
b = torch.zeros(1, requires_grad = True)
b
```

Out[5]:
```
tensor([0.], requires_grad=True)
```

In [6]:
```
h = x_train * W + b
print(h)
```

```
tensor([[0.],
        [0.],
        [0.]], grad_fn=<AddBackward0>)
```

In [7]:
```
cost = torch.mean((h - y_train) ** 2)
cost
```

Out[7]:
```
tensor(18.6667, grad_fn=<MeanBackward0>)
```

In [8]:
```
optimizer = optim.SGD([W, b], lr = 0.01)
```

In [9]:
```
optimizer.zero_grad()

cost.backward()
optimizer.step()
```

In [10]:
```
print(optimizer)
```

```
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.01
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

In [11]:
```
nb_epochs = 2000 # 원하는만큼 경사 하강법을 반복
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    hypothesis = x_train * W + b

    # cost 계산
    cost = torch.mean((hypothesis - y_train) ** 2)

    # cost로 H(x) 개선
    optimizer.zero_grad() # torch는 미분값을 계속 누적시키므로 미분값을 0으로 계속 초기화 시켜줘야 한다.
    cost.backward()   # cost의 미분값 계산
    optimizer.step()

    # 100번마다 로그 출력
    if epoch % 100 == 0:
        print('Epoch {:4d}/{} W: {:.3f}, b: {:.3f} Cost: {:.6f}'.format(
            epoch, nb_epochs, W.item(), b.item(), cost.item()
        ))
```

```
Epoch    0/2000 W: 0.353, b: 0.151 Cost: 14.770963
Epoch  100/2000 W: 1.746, b: 0.577 Cost: 0.047939
Epoch  200/2000 W: 1.801, b: 0.453 Cost: 0.029624
Epoch  300/2000 W: 1.843, b: 0.356 Cost: 0.018306
Epoch  400/2000 W: 1.877, b: 0.280 Cost: 0.011312
Epoch  500/2000 W: 1.903, b: 0.220 Cost: 0.006990
Epoch  600/2000 W: 1.924, b: 0.173 Cost: 0.004319
Epoch  700/2000 W: 1.940, b: 0.136 Cost: 0.002669
Epoch  800/2000 W: 1.953, b: 0.107 Cost: 0.001649
Epoch  900/2000 W: 1.963, b: 0.084 Cost: 0.001019
Epoch 1000/2000 W: 1.971, b: 0.066 Cost: 0.000630
Epoch 1100/2000 W: 1.977, b: 0.052 Cost: 0.000389
Epoch 1200/2000 W: 1.982, b: 0.041 Cost: 0.000240
Epoch 1300/2000 W: 1.986, b: 0.032 Cost: 0.000149
Epoch 1400/2000 W: 1.989, b: 0.025 Cost: 0.000092
Epoch 1500/2000 W: 1.991, b: 0.020 Cost: 0.000057
Epoch 1600/2000 W: 1.993, b: 0.016 Cost: 0.000035
Epoch 1700/2000 W: 1.995, b: 0.012 Cost: 0.000022
Epoch 1800/2000 W: 1.996, b: 0.010 Cost: 0.000013
Epoch 1900/2000 W: 1.997, b: 0.008 Cost: 0.000008
Epoch 2000/2000 W: 1.997, b: 0.006 Cost: 0.000005
```

자동 미분

In [34]:
```
w = torch.tensor(3.0, requires_grad = True)
y = w**2
z = 2*y + 5
w
```

Out[34]: tensor(3., requires_grad=True)

In [31]:
```
z.backward() # z에서 w에 대해 미분수행 => 이 미분값은 .grad 속성에 누적
```

In [35]:
```
print('수식을 w로 미분한 값 : {}'.format(w.grad))
print('마지막 연산: {}'.format(z.grad_fn))
```

```
수식을 w로 미분한 값 : None
마지막 연산: <AddBackward0 object at 0x00000139BCF0B108>
```

다중 선형 회귀

In [13]:
```
x_train  =  torch.FloatTensor([[73,  80,  75],
                               [93,  88,  93],
                               [89,  91,  90],
                               [96,  98,  100],
                               [73,  66,  70]])
y_train  =  torch.FloatTensor([[152],  [185],  [180],  [196],  [142]])

# 모델 초기화
W = torch.zeros((3, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
# optimizer 설정
optimizer = optim.SGD([W, b], lr=1e-5)

nb_epochs = 20
for epoch in range(nb_epochs + 1):

    # H(x) 계산
    # 편향 b는 브로드 캐스팅되어 각 샘플에 더해집니다.
    hypothesis = x_train.matmul(W) + b

    # cost 계산
    cost = torch.mean((hypothesis - y_train) ** 2)

    # cost로 H(x) 개선
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    # 100번마다 로그 출력
    print('Epoch {:4d}/{} hypothesis: {} Cost: {:.6f}'.format(
        epoch, nb_epochs, hypothesis.squeeze().detach(), cost.item()
    ))
```

```
Epoch    0/20 hypothesis: tensor([0., 0., 0., 0., 0.]) Cost: 29661.800781
Epoch    1/20 hypothesis: tensor([67.2578, 80.8397, 79.6523, 86.7394, 61.6605]) Cost: 9298.520508
Epoch    2/20 hypothesis: tensor([104.9128, 126.0990, 124.2466, 135.3015,  96.1821]) Cost: 2915.712402
Epoch    3/20 hypothesis: tensor([125.9942, 151.4381, 149.2133, 162.4896, 115.5097]) Cost: 915.040527
Epoch    4/20 hypothesis: tensor([137.7968, 165.6247, 163.1911, 177.7112, 126.3307]) Cost: 287.936005
Epoch    5/20 hypothesis: tensor([144.4044, 173.5674, 171.0168, 186.2332, 132.3891]) Cost: 91.371010
Epoch    6/20 hypothesis: tensor([148.1035, 178.0144, 175.3980, 191.0042, 135.7812]) Cost: 29.758139
Epoch    7/20 hypothesis: tensor([150.1744, 180.5042, 177.8508, 193.6753, 137.6805]) Cost: 10.445305
Epoch    8/20 hypothesis: tensor([151.3336, 181.8983, 179.2240, 195.1707, 138.7440]) Cost: 4.391228
Epoch    9/20 hypothesis: tensor([151.9824, 182.6789, 179.9928, 196.0079, 139.3396]) Cost: 2.493135
Epoch   10/20 hypothesis: tensor([152.3454, 183.1161, 180.4231, 196.4765, 139.6732]) Cost: 1.897688
Epoch   11/20 hypothesis: tensor([152.5485, 183.3610, 180.6640, 196.7389, 139.8602]) Cost: 1.710541
Epoch   12/20 hypothesis: tensor([152.6620, 183.4982, 180.7988, 196.8857, 139.9651]) Cost: 1.651412
Epoch   13/20 hypothesis: tensor([152.7253, 183.5752, 180.8742, 196.9678, 140.0240]) Cost: 1.632387
Epoch   14/20 hypothesis: tensor([152.7606, 183.6184, 180.9164, 197.0138, 140.0571]) Cost: 1.625923
Epoch   15/20 hypothesis: tensor([152.7802, 183.6427, 180.9399, 197.0395, 140.0759]) Cost: 1.623412
Epoch   16/20 hypothesis: tensor([152.7909, 183.6565, 180.9530, 197.0538, 140.0865]) Cost: 1.622141
Epoch   17/20 hypothesis: tensor([152.7968, 183.6643, 180.9603, 197.0618, 140.0927]) Cost: 1.621253
Epoch   18/20 hypothesis: tensor([152.7999, 183.6688, 180.9644, 197.0662, 140.0963]) Cost: 1.620500
Epoch   19/20 hypothesis: tensor([152.8014, 183.6715, 180.9666, 197.0686, 140.0985]) Cost: 1.619770
Epoch   20/20 hypothesis: tensor([152.8020, 183.6731, 180.9677, 197.0699, 140.1000]) Cost: 1.619033
```

nn module 활용하여 다중 선형 회귀 구현하기

In [47]:
```
x_train  =  torch.FloatTensor([[73,  80,  75],
                               [93,  88,  93],
                               [89,  91,  90],
                               [96,  98,  100],
                               [73,  66,  70]])
y_train  =  torch.FloatTensor([[152],  [185],  [180],  [196],  [142]])
```

In [48]:
```
model = nn.Linear(3,1) # (input 개수, output 개수)
```

In [49]:
```
list(model.parameters())    # random으로 초기화, 첫번째값 : w 두번째값 : b
optimizer = torch.optim.SGD(model.parameters(), lr=1e-6)
```

In [50]:
```
nb_epochs = 2000
for epoch in range(nb_epochs+1):

    # H(x) 계산
    prediction = model(x_train)
    # model(x_train)은 model.forward(x_train)와 동일함.

    # cost 계산
    cost = F.mse_loss(prediction, y_train) # <== 파이토치에서 제공하는 평균 제곱 오차 함수

    # cost로 H(x) 개선하는 부분
    # gradient를 0으로 초기화
    optimizer.zero_grad()
    # 비용 함수를 미분하여 gradient 계산
    cost.backward()
    # W와 b를 업데이트
    optimizer.step()

    if epoch % 100 == 0:
    # 100번마다 로그 출력
      print('Epoch {:4d}/{} Cost: {:.6f}'.format(
          epoch, nb_epochs, cost.item()
      ))
```

```
Epoch    0/2000 Cost: 37019.644531
Epoch  100/2000 Cost: 5.052137
```

```
Epoch  200/2000 Cost: 0.493100
Epoch  300/2000 Cost: 0.490885
Epoch  400/2000 Cost: 0.489245
Epoch  500/2000 Cost: 0.487624
Epoch  600/2000 Cost: 0.485996
Epoch  700/2000 Cost: 0.484392
Epoch  800/2000 Cost: 0.482790
Epoch  900/2000 Cost: 0.481186
Epoch 1000/2000 Cost: 0.479600
Epoch 1100/2000 Cost: 0.478008
Epoch 1200/2000 Cost: 0.476435
Epoch 1300/2000 Cost: 0.474853
Epoch 1400/2000 Cost: 0.473289
Epoch 1500/2000 Cost: 0.471733
Epoch 1600/2000 Cost: 0.470184
Epoch 1700/2000 Cost: 0.468639
Epoch 1800/2000 Cost: 0.467104
Epoch 1900/2000 Cost: 0.465575
Epoch 2000/2000 Cost: 0.464078
```

In [51]:
```
# 임의의 입력 [73, 80, 75]를 선언
new_var =  torch.FloatTensor([[73, 80, 75]])
# 입력한 값 [73, 80, 75]에 대해서 예측값 y를 리턴받아서 pred_y에 저장
pred_y = model(new_var)
print("훈련 후 입력이 73, 80, 75일 때의 예측값 :", pred_y)
```

```
훈련 후 입력이 73, 80, 75일 때의 예측값 : tensor([[150.6435]], grad_fn=<AddmmBackward>)
```

In [52]:
```
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[1.0689, 0.3781, 0.5644]], requires_grad=True), Parameter containing:
tensor([0.0372], requires_grad=True)]
```

class로 모델 구현

In [53]:
```
class MultivariateLinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(3, 1) # 다중 선형 회귀이므로 input_dim=3, output_dim=1.

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

In [56]:
```
model = MultivariateLinearRegressionModel()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5)
```

In [57]:
```
nb_epochs = 2000
for epoch in range(nb_epochs+1):

    # H(x) 계산
    prediction = model(x_train)
    # model(x_train)은 model.forward(x_train)와 동일함.

    # cost 계산
    cost = F.mse_loss(prediction, y_train) # <== 파이토치에서 제공하는 평균 제곱 오차 함수

    # cost로 H(x) 개선하는 부분
    # gradient를 0으로 초기화
    optimizer.zero_grad()
    # 비용 함수를 미분하여 gradient 계산
    cost.backward()
    # W와 b를 업데이트
    optimizer.step()

    if epoch % 100 == 0:
    # 100번마다 로그 출력
      print('Epoch {:4d}/{} Cost: {:.6f}'.format(
          epoch, nb_epochs, cost.item()
      ))
```

```
Epoch    0/2000 Cost: 74817.734375
Epoch  100/2000 Cost: 1.783027
Epoch  200/2000 Cost: 1.702140
Epoch  300/2000 Cost: 1.625501
Epoch  400/2000 Cost: 1.552908
Epoch  500/2000 Cost: 1.484089
Epoch  600/2000 Cost: 1.418908
Epoch  700/2000 Cost: 1.357123
Epoch  800/2000 Cost: 1.298609
Epoch  900/2000 Cost: 1.243148
Epoch 1000/2000 Cost: 1.190584
Epoch 1100/2000 Cost: 1.140777
Epoch 1200/2000 Cost: 1.093576
Epoch 1300/2000 Cost: 1.048849
Epoch 1400/2000 Cost: 1.006471
Epoch 1500/2000 Cost: 0.966296
Epoch 1600/2000 Cost: 0.928213
Epoch 1700/2000 Cost: 0.892132
Epoch 1800/2000 Cost: 0.857937
Epoch 1900/2000 Cost: 0.825518
Epoch 2000/2000 Cost: 0.794799
```

미니배치와 데이터 로드

In [58]:
```
x_train = torch.FloatTensor([[73, 80, 75],
                             [93, 88, 93],
                             [89, 91, 90],
                             [96, 98, 100],
                             [73, 66, 70]])
y_train = torch.FloatTensor([[152], [185], [180], [196], [142]])
```

In [59]:
```
from torch.utils.data import TensorDataset # 텐서데이터셋
from torch.utils.data import DataLoader # 데이터로더
```

In [60]:
```
dataset = TensorDataset(x_train, y_train)
```

In [61]:
```
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)
```

In [62]:
```
model = nn.Linear(3,1)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5)
```

In [65]:
```
nb_epochs = 20
for epoch in range(nb_epochs + 1):
    for batch_idx, samples in enumerate(dataloader):
        #print(batch_idx)
        print(samples)
        x_train, y_train = samples
        # H(x) 계산
        prediction = model(x_train)

        # cost 계산
        cost = F.mse_loss(prediction, y_train)

        # cost로 H(x) 계산
        optimizer.zero_grad()
        cost.backward()
        optimizer.step()

    print('Epoch {:4d}/{} Batch {}/{} Cost: {:.6f}'.format(
        epoch, nb_epochs, batch_idx+1, len(dataloader),
        cost.item()
        ))
```

```
[tensor([[73., 66., 70.],
        [93., 88., 93.]]), tensor([[142.],
        [185.]])]
Epoch    0/20 Batch 1/3 Cost: 2.584844
[tensor([[ 96.,  98., 100.],
        [ 89.,  91.,  90.]]), tensor([[196.],
        [180.]])]
Epoch    0/20 Batch 2/3 Cost: 4.842172
[tensor([[73., 80., 75.]]), tensor([[152.]])]
Epoch    0/20 Batch 3/3 Cost: 0.000537
[tensor([[ 73.,  66.,  70.],
        [ 96.,  98., 100.]]), tensor([[142.],
        [196.]])]
Epoch    1/20 Batch 1/3 Cost: 4.418236
[tensor([[73., 80., 75.],
        [93., 88., 93.]]), tensor([[152.],
        [185.]])]
Epoch    1/20 Batch 2/3 Cost: 0.745866
[tensor([[89., 91., 90.]]), tensor([[180.]])]
Epoch    1/20 Batch 3/3 Cost: 0.464696
[tensor([[ 96.,  98., 100.],
        [ 73.,  80.,  75.]]), tensor([[196.],
        [152.]])]
Epoch    2/20 Batch 1/3 Cost: 1.314902
[tensor([[73., 66., 70.],
        [89., 91., 90.]]), tensor([[142.],
        [180.]])]
Epoch    2/20 Batch 2/3 Cost: 3.979614
[tensor([[93., 88., 93.]]), tensor([[185.]])]
Epoch    2/20 Batch 3/3 Cost: 1.280396
[tensor([[89., 91., 90.],
        [73., 66., 70.]]), tensor([[180.],
        [142.]])]
Epoch    3/20 Batch 1/3 Cost: 2.455002
[tensor([[93., 88., 93.],
        [73., 80., 75.]]), tensor([[185.],
        [152.]])]
Epoch    3/20 Batch 2/3 Cost: 0.318939
[tensor([[ 96.,  98., 100.]]), tensor([[196.]])]
Epoch    3/20 Batch 3/3 Cost: 6.060345
[tensor([[73., 66., 70.],
        [89., 91., 90.]]), tensor([[142.],
        [180.]])]
Epoch    4/20 Batch 1/3 Cost: 4.215952
[tensor([[ 93.,  88.,  93.],
        [ 96.,  98., 100.]]), tensor([[185.],
        [196.]])]
Epoch    4/20 Batch 2/3 Cost: 2.131625
[tensor([[73., 80., 75.]]), tensor([[152.]])]
Epoch    4/20 Batch 3/3 Cost: 0.008160
[tensor([[ 73.,  80.,  75.],
        [ 96.,  98., 100.]]), tensor([[152.],
        [196.]])]
Epoch    5/20 Batch 1/3 Cost: 1.218369
[tensor([[89., 91., 90.],
        [93., 88., 93.]]), tensor([[180.],
        [185.]])]
Epoch    5/20 Batch 2/3 Cost: 1.479177
[tensor([[73., 66., 70.]]), tensor([[142.]])]
Epoch    5/20 Batch 3/3 Cost: 6.206132
[tensor([[ 96.,  98., 100.],
        [ 93.,  88.,  93.]]), tensor([[196.],
        [185.]])]
Epoch    6/20 Batch 1/3 Cost: 3.513988
[tensor([[73., 66., 70.],
        [73., 80., 75.]]), tensor([[142.],
        [152.]])]
Epoch    6/20 Batch 2/3 Cost: 2.550050
[tensor([[89., 91., 90.]]), tensor([[180.]])]
Epoch    6/20 Batch 3/3 Cost: 1.060363
[tensor([[73., 66., 70.],
        [73., 80., 75.]]), tensor([[142.],
        [152.]])]
Epoch    7/20 Batch 1/3 Cost: 2.756344
[tensor([[ 96.,  98., 100.],
        [ 93.,  88.,  93.]]), tensor([[196.],
```

```
        [185.]])]
Epoch    7/20 Batch 2/3 Cost: 2.776859
[tensor([[89., 91., 90.]]), tensor([[180.]])]
Epoch    7/20 Batch 3/3 Cost: 0.252263
[tensor([[73., 66., 70.],
        [73., 80., 75.]]), tensor([[142.],
        [152.]])]
Epoch    8/20 Batch 1/3 Cost: 3.261924
[tensor([[ 96.,  98., 100.],
        [ 89.,  91.,  90.]]), tensor([[196.],
        [180.]])]
Epoch    8/20 Batch 2/3 Cost: 2.416992
[tensor([[93., 88., 93.]]), tensor([[185.]])]
Epoch    8/20 Batch 3/3 Cost: 2.486579
[tensor([[73., 80., 75.],
        [89., 91., 90.]]), tensor([[152.],
        [180.]])]
Epoch    9/20 Batch 1/3 Cost: 0.391344
[tensor([[73., 66., 70.],
        [93., 88., 93.]]), tensor([[142.],
        [185.]])]
Epoch    9/20 Batch 2/3 Cost: 3.296580
[tensor([[ 96.,  98., 100.]]), tensor([[196.]])]
Epoch    9/20 Batch 3/3 Cost: 6.679893
[tensor([[89., 91., 90.],
        [93., 88., 93.]]), tensor([[180.],
        [185.]])]
Epoch   10/20 Batch 1/3 Cost: 1.518231
[tensor([[ 73.,  80.,  75.],
        [ 96.,  98., 100.]]), tensor([[152.],
        [196.]])]
Epoch   10/20 Batch 2/3 Cost: 1.275168
[tensor([[73., 66., 70.]]), tensor([[142.]])]
Epoch   10/20 Batch 3/3 Cost: 7.933348
[tensor([[ 96.,  98., 100.],
        [ 73.,  66.,  70.]]), tensor([[196.],
        [142.]])]
Epoch   11/20 Batch 1/3 Cost: 4.629972
[tensor([[93., 88., 93.],
        [89., 91., 90.]]), tensor([[185.],
        [180.]])]
Epoch   11/20 Batch 2/3 Cost: 0.634281
[tensor([[73., 80., 75.]]), tensor([[152.]])]
Epoch   11/20 Batch 3/3 Cost: 0.112639
[tensor([[73., 66., 70.],
        [89., 91., 90.]]), tensor([[142.],
        [180.]])]
Epoch   12/20 Batch 1/3 Cost: 2.746875
[tensor([[93., 88., 93.],
        [73., 80., 75.]]), tensor([[185.],
        [152.]])]
Epoch   12/20 Batch 2/3 Cost: 0.333674
[tensor([[ 96.,  98., 100.]]), tensor([[196.]])]
Epoch   12/20 Batch 3/3 Cost: 5.316227
[tensor([[ 96.,  98., 100.],
        [ 73.,  80.,  75.]]), tensor([[196.],
        [152.]])]
Epoch   13/20 Batch 1/3 Cost: 0.611344
[tensor([[73., 66., 70.],
        [89., 91., 90.]]), tensor([[142.],
        [180.]])]
Epoch   13/20 Batch 2/3 Cost: 4.726272
[tensor([[93., 88., 93.]]), tensor([[185.]])]
Epoch   13/20 Batch 3/3 Cost: 1.738031
[tensor([[ 96.,  98., 100.],
        [ 73.,  66.,  70.]]), tensor([[196.],
        [142.]])]
Epoch   14/20 Batch 1/3 Cost: 4.579443
[tensor([[89., 91., 90.],
        [93., 88., 93.]]), tensor([[180.],
        [185.]])]
Epoch   14/20 Batch 2/3 Cost: 0.635563
[tensor([[73., 80., 75.]]), tensor([[152.]])]
Epoch   14/20 Batch 3/3 Cost: 0.103109
[tensor([[ 96.,  98., 100.],
        [ 93.,  88.,  93.]]), tensor([[196.],
        [185.]])]
Epoch   15/20 Batch 1/3 Cost: 2.303972
[tensor([[73., 80., 75.],
        [73., 66., 70.]]), tensor([[152.],
        [142.]])]
Epoch   15/20 Batch 2/3 Cost: 3.046363
[tensor([[89., 91., 90.]]), tensor([[180.]])]
Epoch   15/20 Batch 3/3 Cost: 0.673639
[tensor([[89., 91., 90.],
        [73., 80., 75.]]), tensor([[180.],
        [152.]])]
Epoch   16/20 Batch 1/3 Cost: 0.089745
[tensor([[93., 88., 93.],
        [73., 66., 70.]]), tensor([[185.],
        [142.]])]
Epoch   16/20 Batch 2/3 Cost: 3.942426
[tensor([[ 96.,  98., 100.]]), tensor([[196.]])]
Epoch   16/20 Batch 3/3 Cost: 5.924904
[tensor([[89., 91., 90.],
        [73., 80., 75.]]), tensor([[180.],
        [152.]])]
Epoch   17/20 Batch 1/3 Cost: 0.133270
[tensor([[ 96.,  98., 100.],
        [ 73.,  66.,  70.]]), tensor([[196.],
        [142.]])]
Epoch   17/20 Batch 2/3 Cost: 4.557955
[tensor([[93., 88., 93.]]), tensor([[185.]])]
Epoch   17/20 Batch 3/3 Cost: 2.046625
```

```
[tensor([[73., 80., 75.],
        [73., 66., 70.]]), tensor([[152.],
        [142.]])]
Epoch   18/20 Batch 1/3 Cost: 2.221971
[tensor([[89., 91., 90.],
        [93., 88., 93.]]), tensor([[180.],
        [185.]])]
Epoch   18/20 Batch 2/3 Cost: 0.766555
[tensor([[ 96.,  98., 100.]]), tensor([[196.]])]
Epoch   18/20 Batch 3/3 Cost: 5.349986
[tensor([[ 96.,  98., 100.],
        [ 73.,  80.,  75.]]), tensor([[196.],
        [152.]])]
Epoch   19/20 Batch 1/3 Cost: 0.615533
[tensor([[73., 66., 70.],
        [93., 88., 93.]]), tensor([[142.],
        [185.]])]
Epoch   19/20 Batch 2/3 Cost: 6.596270
[tensor([[89., 91., 90.]]), tensor([[180.]])]
Epoch   19/20 Batch 3/3 Cost: 0.450125
[tensor([[73., 80., 75.],
        [73., 66., 70.]]), tensor([[152.],
        [142.]])]
Epoch   20/20 Batch 1/3 Cost: 3.046163
[tensor([[93., 88., 93.],
        [89., 91., 90.]]), tensor([[185.],
        [180.]])]
Epoch   20/20 Batch 2/3 Cost: 0.623378
[tensor([[ 96.,  98., 100.]]), tensor([[196.]])]
Epoch   20/20 Batch 3/3 Cost: 4.478866
```

In [66]:
```
# 임의의 입력 [73, 80, 75]를 선언
new_var =  torch.FloatTensor([[73, 80, 75]])
# 입력한 값 [73, 80, 75]에 대해서 예측값 y를 리턴받아서 pred_y에 저장
pred_y = model(new_var)
print("훈련 후 입력이 73, 80, 75일 때의 예측값 :", pred_y)
```

```
훈련 후 입력이 73, 80, 75일 때의 예측값 : tensor([[151.4131]], grad_fn=<AddmmBackward>)
```

커스텀 데이터셋

torch.utils.data.Dataset을 상속받아 직접 커스텀 데이터셋(Custom Dataset)을 만드는 경우

class CustomDataset(torch.utils.data.Dataset): def **init**(self): 데이터셋의 전처리를 해주는 부분

def **len**(self): 데이터셋의 길이. 즉, 총 샘플의 수를 적어주는 부분

def **getitem**(self, idx): 데이터셋에서 특정 1개의 샘플을 가져오는 함수

In [69]:
```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
```

In [70]:
```
# Dataset 상속
class CustomDataset(Dataset):
  def __init__(self):
    self.x_data = [[73, 80, 75],
                   [93, 88, 93],
                   [89, 91, 90],
                   [96, 98, 100],
                   [73, 66, 70]]
    self.y_data = [[152], [185], [180], [196], [142]]

  # 총 데이터의 개수를 리턴
  def __len__(self):
    return len(self.x_data)

  # 인덱스를 입력받아 그에 맵핑되는 입출력 데이터를 파이토치의 Tensor 형태로 리턴
  def __getitem__(self, idx):
    x = torch.FloatTensor(self.x_data[idx])
    y = torch.FloatTensor(self.y_data[idx])
    return x, y
```

In [71]:
```
dataset = CustomDataset()
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)
```

In [72]:
```
model = torch.nn.Linear(3,1)
optimizer = torch.optim.SGD(model.parameters(), lr=1e-5)
```

In [73]:
```
nb_epochs = 20
for epoch in range(nb_epochs + 1):
  for batch_idx, samples in enumerate(dataloader):
    # print(batch_idx)
    # print(samples)
    x_train, y_train = samples
    # H(x) 계산
    prediction = model(x_train)

    # cost 계산
    cost = F.mse_loss(prediction, y_train)

    # cost로 H(x) 계산
    optimizer.zero_grad()
    cost.backward()
    optimizer.step()

    print('Epoch {:4d}/{} Batch {}/{} Cost: {:.6f}'.format(
        epoch, nb_epochs, batch_idx+1, len(dataloader),
        cost.item()
        ))
```

```
Epoch    0/20 Batch 1/3 Cost: 13731.738281
Epoch    0/20 Batch 2/3 Cost: 10536.747070
Epoch    0/20 Batch 3/3 Cost: 1879.159546
Epoch    1/20 Batch 1/3 Cost: 527.531494
```

```
Epoch    1/20 Batch 2/3 Cost: 117.137711
Epoch    1/20 Batch 3/3 Cost: 144.328674
Epoch    2/20 Batch 1/3 Cost: 18.158066
Epoch    2/20 Batch 2/3 Cost: 20.904289
Epoch    2/20 Batch 3/3 Cost: 0.877877
Epoch    3/20 Batch 1/3 Cost: 19.135265
Epoch    3/20 Batch 2/3 Cost: 4.632922
Epoch    3/20 Batch 3/3 Cost: 17.950356
Epoch    4/20 Batch 1/3 Cost: 9.492016
Epoch    4/20 Batch 2/3 Cost: 22.948545
Epoch    4/20 Batch 3/3 Cost: 17.450277
Epoch    5/20 Batch 1/3 Cost: 18.712929
Epoch    5/20 Batch 2/3 Cost: 8.011886
Epoch    5/20 Batch 3/3 Cost: 2.653684
Epoch    6/20 Batch 1/3 Cost: 7.324215
Epoch    6/20 Batch 2/3 Cost: 16.822460
Epoch    6/20 Batch 3/3 Cost: 10.819638
Epoch    7/20 Batch 1/3 Cost: 9.476420
Epoch    7/20 Batch 2/3 Cost: 18.761503
Epoch    7/20 Batch 3/3 Cost: 2.511521
Epoch    8/20 Batch 1/3 Cost: 8.723792
Epoch    8/20 Batch 2/3 Cost: 15.299099
Epoch    8/20 Batch 3/3 Cost: 11.549377
Epoch    9/20 Batch 1/3 Cost: 3.692261
Epoch    9/20 Batch 2/3 Cost: 11.960111
Epoch    9/20 Batch 3/3 Cost: 31.345032
Epoch   10/20 Batch 1/3 Cost: 8.905148
Epoch   10/20 Batch 2/3 Cost: 29.020100
Epoch   10/20 Batch 3/3 Cost: 2.697261
Epoch   11/20 Batch 1/3 Cost: 16.551226
Epoch   11/20 Batch 2/3 Cost: 9.007263
Epoch   11/20 Batch 3/3 Cost: 18.028404
Epoch   12/20 Batch 1/3 Cost: 9.328495
Epoch   12/20 Batch 2/3 Cost: 13.225152
Epoch   12/20 Batch 3/3 Cost: 13.593267
Epoch   13/20 Batch 1/3 Cost: 9.183700
Epoch   13/20 Batch 2/3 Cost: 15.547651
Epoch   13/20 Batch 3/3 Cost: 11.002498
Epoch   14/20 Batch 1/3 Cost: 25.680096
Epoch   14/20 Batch 2/3 Cost: 11.227829
Epoch   14/20 Batch 3/3 Cost: 5.011064
Epoch   15/20 Batch 1/3 Cost: 20.490099
Epoch   15/20 Batch 2/3 Cost: 9.635252
Epoch   15/20 Batch 3/3 Cost: 1.706433
Epoch   16/20 Batch 1/3 Cost: 17.064608
Epoch   16/20 Batch 2/3 Cost: 8.984585
Epoch   16/20 Batch 3/3 Cost: 7.809371
Epoch   17/20 Batch 1/3 Cost: 2.499519
Epoch   17/20 Batch 2/3 Cost: 12.834615
Epoch   17/20 Batch 3/3 Cost: 31.847988
Epoch   18/20 Batch 1/3 Cost: 5.607058
Epoch   18/20 Batch 2/3 Cost: 15.083270
Epoch   18/20 Batch 3/3 Cost: 32.278091
Epoch   19/20 Batch 1/3 Cost: 10.024024
Epoch   19/20 Batch 2/3 Cost: 12.710304
Epoch   19/20 Batch 3/3 Cost: 15.130196
Epoch   20/20 Batch 1/3 Cost: 7.753523
Epoch   20/20 Batch 2/3 Cost: 16.647579
Epoch   20/20 Batch 3/3 Cost: 10.140127
```

In [74]:
```python
# 임의의 입력 [73, 80, 75]를 선언
new_var =  torch.FloatTensor([[73, 80, 75]])
# 입력한 값 [73, 80, 75]에 대해서 예측값 y를 리턴받아서 pred_y에 저장
pred_y = model(new_var)
print("훈련 후 입력이 73, 80, 75일 때의 예측값 :", pred_y)
```

```
훈련 후 입력이 73, 80, 75일 때의 예측값 : tensor([[154.0791]], grad_fn=<AddmmBackward>)
```