week6_Chapter3 처음 시작하는 머신 러닝

0. 경사하강법이란?(GD)

- 개요
  - 머신 러닝 및 최적화 문제에서 널리 사용되는 알고리즘
  - 함수 최댓값 or 최솟값 찾는 방법
  - 딥러닝에서 비용 함수 또는 손실 함수 (loss function)가 최소화되는 지점을 찾아 모델 학습시키는 데 사용
- 과정

1. **예측계산(예측함수)**

$$Yp = W * X + B$$

입력텐서 X가 들어가면 예측 결과로 출력 텐서 Yp가 나옴. 즉, 예측함수를 통해 예측값으로 Yp를 구하는 것을 **"예측계산"**이라 한다.

W, B 값에 결과가 달라진다.

2. **손실 계산**

예측 계산 결과 Yp와 정답 Y 값의 차이 = 손실.

머신러닝에서는 **손실이 가장 작아지게 하는 파라미터 W,B를 계산 하는 과정을 "손실 계산"**이라 한다.

보통 손실 함수에서는 모든 데이터 계열의 차이를 제곱해서 그 평균을 계산하는 **평균제곱오차(MSE) 함수**를 사용한다.

3. 경사 계산

W, B의 값을 바꿔가면서 변화한 손실의 정도(경사)를 살피는 과정을 "경사 계산"이라 한다. 손실 함수의 최저값을 목표로 하는 W,B를 옮길 가장 좋은 방향이 경사에 해당된다.

4. 파라미터 수정

경사값에 학습률(lr)을 곱해 그만큼 W,B를 동시에 줄여나가는 과정을 “파라미터 수정”이라 한다.

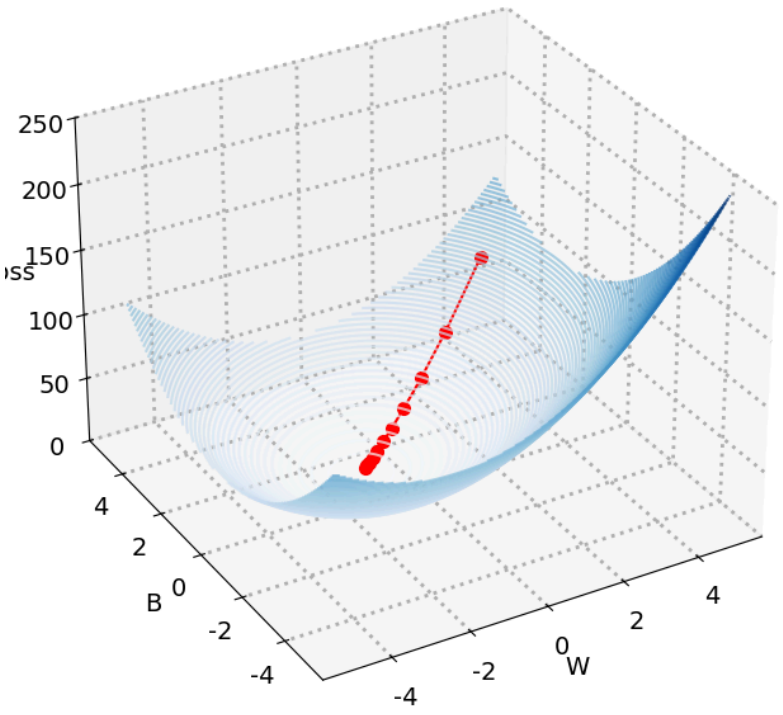
학습률이 너무 크면 최적점을 지나쳐서 왔다 갔다 하며 발산 (Divergence) 손실(loss)이 줄어들지 않거나 오히려 커질 수도 있으며 학습이 불안정해지고, 모델이 수렴하지 않는 overshooting 발생할 수 있다.

학습률이 너무 작으면 수렴 속도가 매우 느려지고, 최적점에 도달하는 데 시간이 오래 걸리며 지역 최저점(local minimum)에 갇힐 가능성이 높다.

1. GD 3D plot

```
def L(u, v):
    return 3 * u**2 + 3 * v**2 - u*v + 7*u - 7*v + 10
def Lu(u, v):
    return 6* u - v + 7
def Lv(u, v):
    return 6* v - u - 7

u = np.linspace(-5, 5, 501)
v = np.linspace(-5, 5, 501)
U, V = np.meshgrid(u, v)
Z = L(U, V)

# 경사 하강법 시뮬레이션
W = np.array([4.0, 4.0])
W1 = [W[0]]
W2 = [W[1]]
N = 21
alpha = 0.05
for i in range(N):
    W = W - alpha *np.array([Lu(W[0], W[1]), Lv(W[0], W[1])])
    W1.append(W[0])
    W2.append(W[1])

n_loop=11

WW1 = np.array(W1[:n_loop])
WW2 = np.array(W2[:n_loop])
ZZ = L(WW1, WW2)
```

```
fig = plt.figure(figsize=(8,8))
ax = plt.axes(projection='3d')
ax.set_zlim(0,250)
ax.set_xlabel('W')
ax.set_ylabel('B')
ax.set_zlabel('loss')
ax.view_init(30, 240)
ax.xaxis._axinfo["grid"]['linewidth'] = 2.
ax.yaxis._axinfo["grid"]['linewidth'] = 2.
ax.zaxis._axinfo["grid"]['linewidth'] = 2.
ax.contour3D(U, V, Z, 100, cmap='Blues', alpha=0.7)
ax.plot3D(WW1, WW2, ZZ, 'o-', c='r', alpha=1, markersize=7)
plt.show()
```



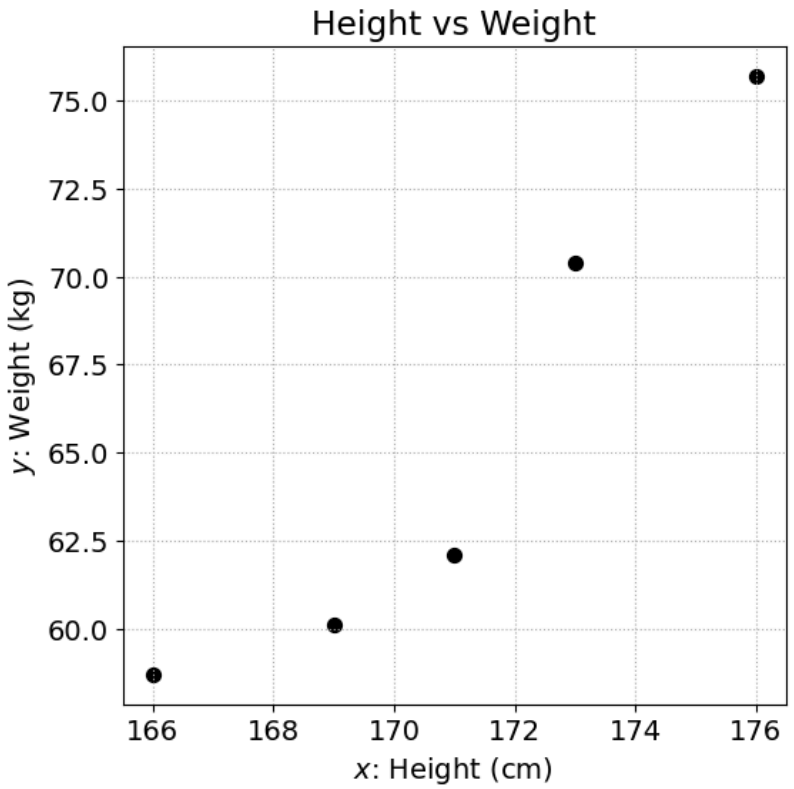
<경사 하강법 구현 이미지>

2. 데이터 전처리

다섯명의 신장과 체중 데이터를 사용하며 1차 함수를 사용해 신장으로 체중을 예측하는 경우, 최적 직선을 구하려고 하는 것이 목적.

```
# 샘플 데이터 선언
sampleData1 = np.array([
    [166, 58.7],
    [176.0, 75.7],
    [171.0, 62.1],
    [173.0, 70.4],
    [169.0,60.1]
])
print(sampleData1)

# 머신러닝 모델에서 사용하기 위해, 신장을 변수 x로,
# 체중을 변수 y로 함

x = sampleData1[:,0]    #height
y = sampleData1[:,1]    #weight

# 산포도 출력 확인

plt.scatter(x,  y,  c='k',  s=50)
# plt.plot([166, 176], [60, 75], 'r:')
plt.xlabel('$x$: Height (cm) ')
plt.ylabel('$y$: Weight (kg)')
plt.title('Height vs Weight')
plt.show()
```

첫 번째 전처리로 학습 데이터를 입력 데이터 x와 정답 데이터 y로 분할.

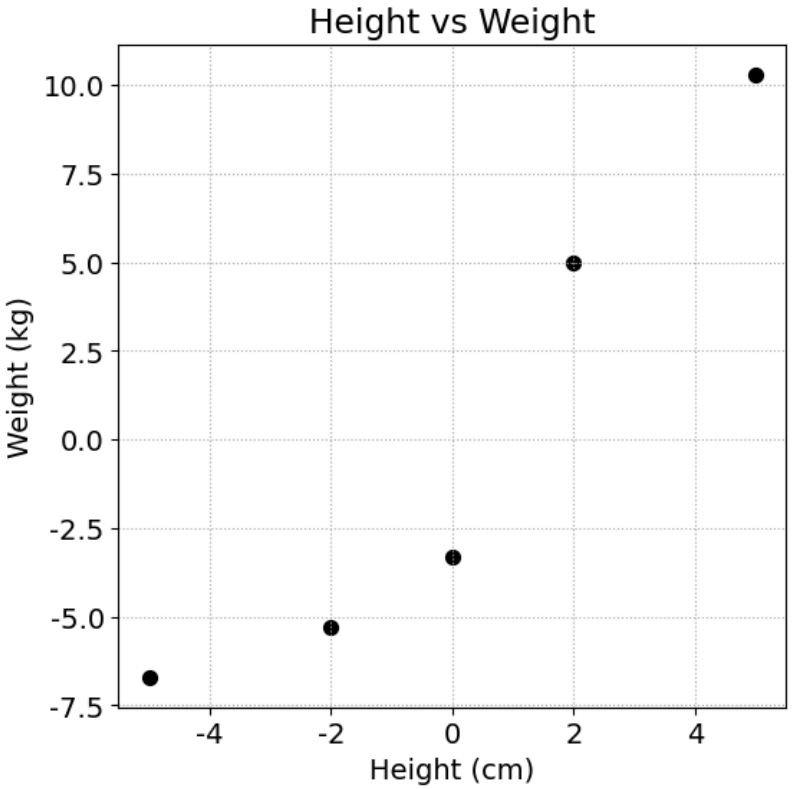


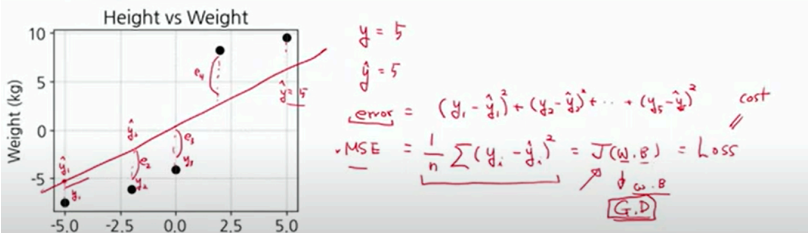
3. 데이터 변환

ML 모델에서 데이터는 0에 가까운 값을 갖는 것이 바람직하기에 x,y 평균값이 0이 되도록 평행이동 시켜서 새로운 좌표계를 X,Y로 설정.

각 평균값을 데이터에서 빼면 경사 하강법이 훨씬 수월해짐.

```
X = x - x.mean()
Y = y - y.mean()

# 산포도를 통해 결과 확인

plt.scatter(X,  Y,  c='k',  s=50)
plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')
plt.title('Height vs Weight')
plt.show()
```





4. 예측 계산

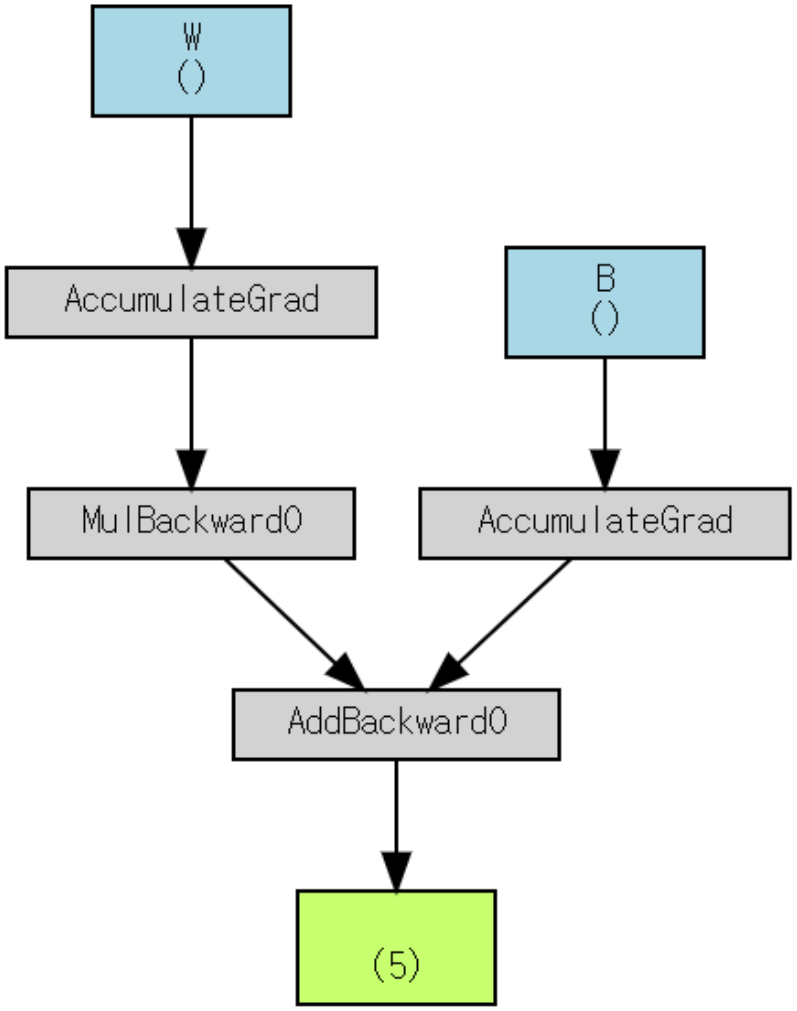
변환한 데이터 X,Y를 텐서 변수로 바꾼 이후 W, B도 텐서 변수로 정의한다.

```
# X와 Y를 텐서 변수로 변환

X = torch.tensor(X).float()
Y = torch.tensor(Y).float()

print(X.dtype)
print(Y.dtype)
#torch.float32
#torch.float32
# 결과 확인
```

```
print(X)
print(Y)
#tensor([-5.,  5.,  0.,  2., -2.])
#tensor([-6.7000, 10.3000, -3.3000,  5.0000, -5.3000])
```

초기값을 1.0으로 설정하고 자동 미분이 가능하도록 설정한다.

```
# 파라미터 정의
# W와 B는 경사 계산을 위해, requires_grad=True 로 설정함

W = torch.tensor(1.0, requires_grad=True).float()
B = torch.tensor(1.0, requires_grad=True).float()

# 예측 함수는 1차 함수(perceptron)

def pred(X): #퍼셉트론
    return W * X + B #W=1, B=1이기에 초기값

# 예측 값 계산
Yp =  pred(X)
# 결과 확인
print(Yp)
#tensor([-4.,  6.,  1.,  3., -1.], grad_fn=<AddBackward0>)
# 예측 값의 계산 그래프 표시
params = {'W': W, 'B': B}
g = make_dot(Yp, params=params)
display(g)
```

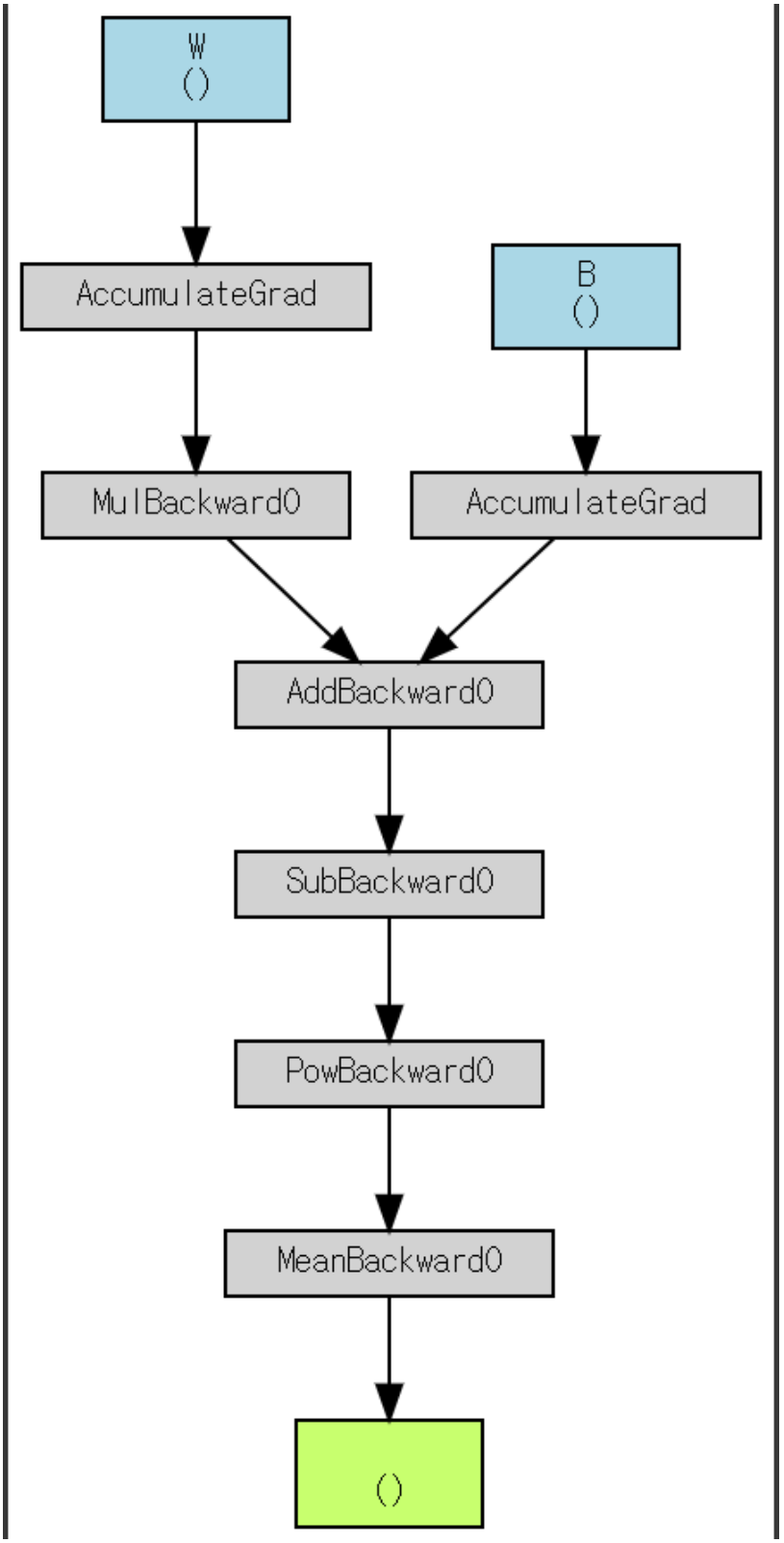


5. 손실계산(MSE)

- 평균 제곱 오차 손실 함수(Loss function)

```
#손실함수, Loss function/ Cost function
##mse loss
def mse(Yp, Y):
    loss = ((Yp - Y) ** 2).mean()
    return loss

# 손실 계산
loss = mse(Yp, Y)

# 결과 표시
print(loss) #여기서 loss는 1차 함수 계수인 W, 정수항 B의 함수
```

```
#tensor(13.3520, grad_fn=<MeanBackward0>)
# 손실 계산 그래프 출력

params = {'W': W, 'B': B}
g = make_dot(loss, params=params)
display(g)
```



여기서 손실은 '예측 함수'와 '손실 함수'의 합성 함수.

즉, 손실은 예측 함수의 거동을 결정 짓는 파라미터(W,B)의 함수.

6. 경사 계산

```
# 경사 계산

loss.backward() #모든 점에서 미분 값
# loss.backward(retain_graph=True)

# 경삿값 확인
print(W.grad)
print(B.grad)
#tensor(-19.0400) #W=1에서의 기울기 값
#tensor(2.0000)
```

7. 파라미터 수정(학습)

W,B는 계산이 한번 끝났기 때문에 갱신이 불가능한 상태임. 그래서 수정이 필요함.

lr은 보통 0.01이나 0.001 로 설정하는 것이 일반적.

```
# 학습률 정의

lr = 0.001

#  경사를 기반으로 파라미터 수정
# W -= lr * W.grad
# B -= lr * B.grad ## Error

#올바른 파라미터 수정 방법
with torch.no_grad():
    W -= lr * W.grad #기울기 방향으로 W를 업데이트
    B -= lr * B.grad #기울기 방향으로 B를 업데이트

    # 계산이 끝난 경삿값을 초기화함
    #pytorch에서 .backward() 호출할 때마다 기울기 누적
    #되기 때문에 초기화 해줘야 함
    W.grad.zero_()
    B.grad.zero_()

# 파라미터 경삿값 확인
```

```
print(W) #학습 진행하면서 값 update
print(B)
print(W.grad) #0으로 초기화.
print(B.grad)
#tensor(1.0190, requires_grad=True)
#tensor(0.9980, requires_grad=True)
#tensor(0.)
#tensor(0.)
```

W, B의 초기값은 모두 1.0 이었으므로, W는 증가하고 B는 감소하는 방향으로 변화함 ⇒ 반복해서 최적의 값으로 수렴시키는 것이 경사 하강법.

8. 반복계산

```
# 초기화

# W와 B를 변수로 사용
W = torch.tensor(1.0, requires_grad=True).float()
B = torch.tensor(1.0, requires_grad=True).float()

# 반복 횟수
num_epochs = 500

# 학습률
lr = 0.001

# history 기록을 위한 배열 초기화
history = np.zeros((0, 2))
```

```
# 루프 처리

for epoch in range(num_epochs):

    # 예측 계산
    Yp = pred(X)

    # 손실 계산
```

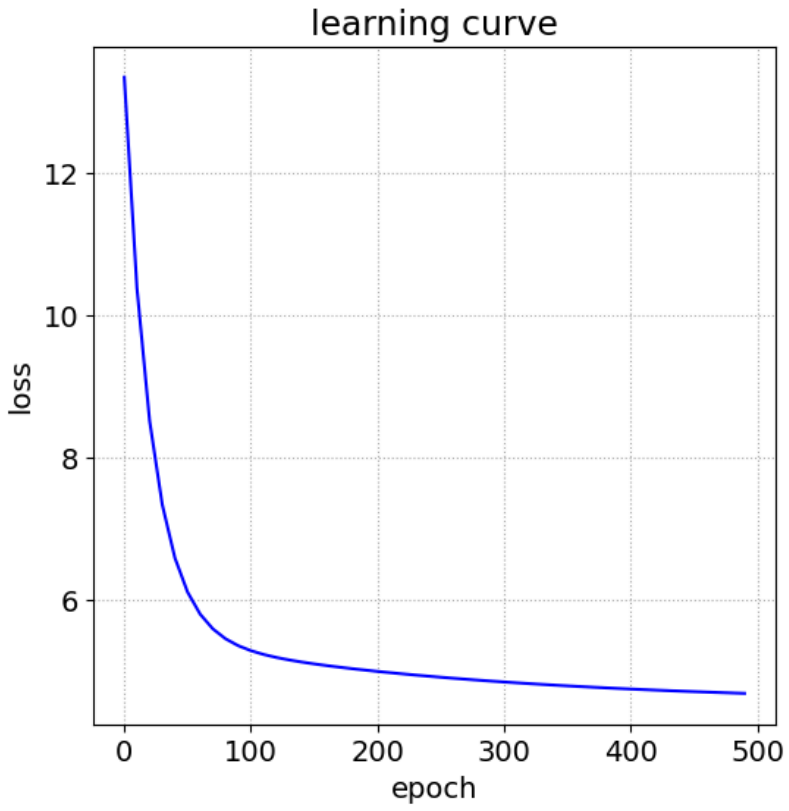
```
    loss = mse(Yp, Y) #torch.tensor

    # 경사 계산
    loss.backward()

    with torch.no_grad():
        # 파라미터 수정
        W -= lr * W.grad
        B -= lr * B.grad

        # 경삿값 초기화
        W.grad.zero_()
        B.grad.zero_()

    # 손실 기록
    # 몇 회 째의 반복인지와 그때 손실값 출력
    if (epoch %10 == 0):
        item = np.array([epoch, loss.item()]) #loss가 주는 방향으로 가야 하기에 loss
        history = np.vstack((history, item)) #vertical 방향으로 item stack.
        print(f'epoch = {epoch}  loss = {loss:.4f}')

#결과
#epoch = 0  loss = 13.3520
#epoch = 10  loss = 10.3855
#epoch = 20  loss = 8.5173
#...
#epoch = 490  loss = 4.6796
```

9. 결과 평가

W, B의 최종 결과값과 시작과 종료 시점의 손실 값을 확인한다.

```
# 최종 파라미터 값
print('W = ', W.data.numpy())
print('B = ', B.data.numpy())

# 손실 확인
```

```
print(f'초기상태 : 손실:{history[0,1]:.4f}')
print(f'최종상태 : 손실:{history[-1,1]:.4f}')
#W =  1.820683
#B =  0.3675114
#초기상태 : 손실:13.3520
#최종상태 : 손실:4.6796

# 학습 곡선 출력(손실)

plt.plot(history[:,0], history[:,1], 'b')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('learning curve')
plt.show()
```

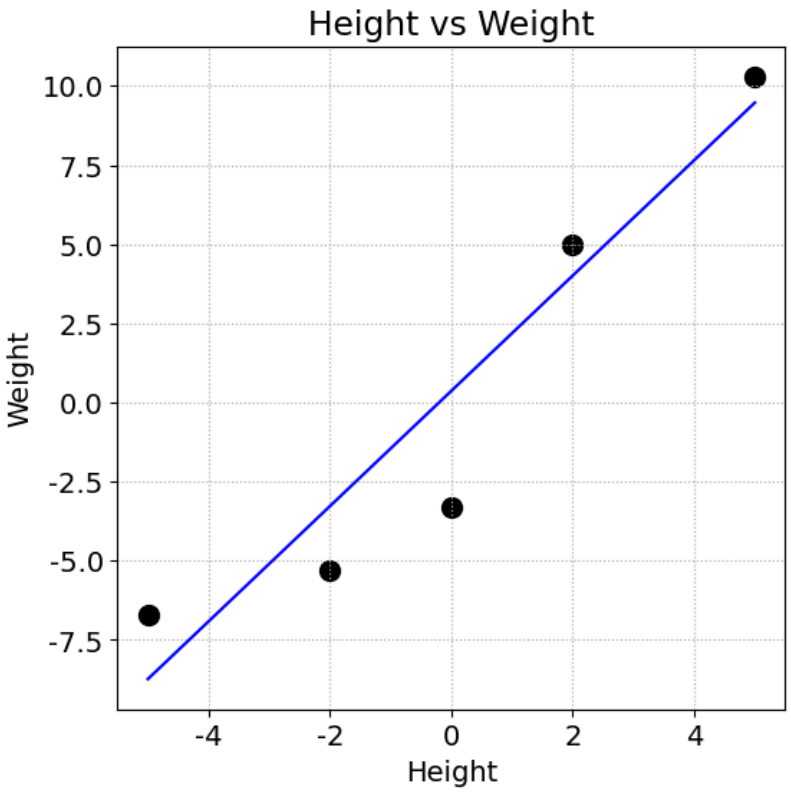


10. 산포도에 회귀 직선 동시 출력

반복 계산을 통해 구한 W,B 값으로부터 직선의 식 산출하고 산포도에 겹쳐 그린다.

```
print(W.item())
print(B.item())
#1.8206830024719238
#0.3675113916397095

W_numpy = W.item() #W의 기울기
B_numpy = B.item()

# x의 범위를 구함(Xrange)
X_max = X.max()
X_min = X.min()
X_range = np.array((X_min, X_max))
X_range = torch.from_numpy(X_range).float()
print(X_range)
#tensor([-5.,  5.])

# 이와 대응하는 예측값 y를 구함
Y_range = pred(X_range)
print(Y_range.data)
#tensor([-8.7359,  9.4709])
```

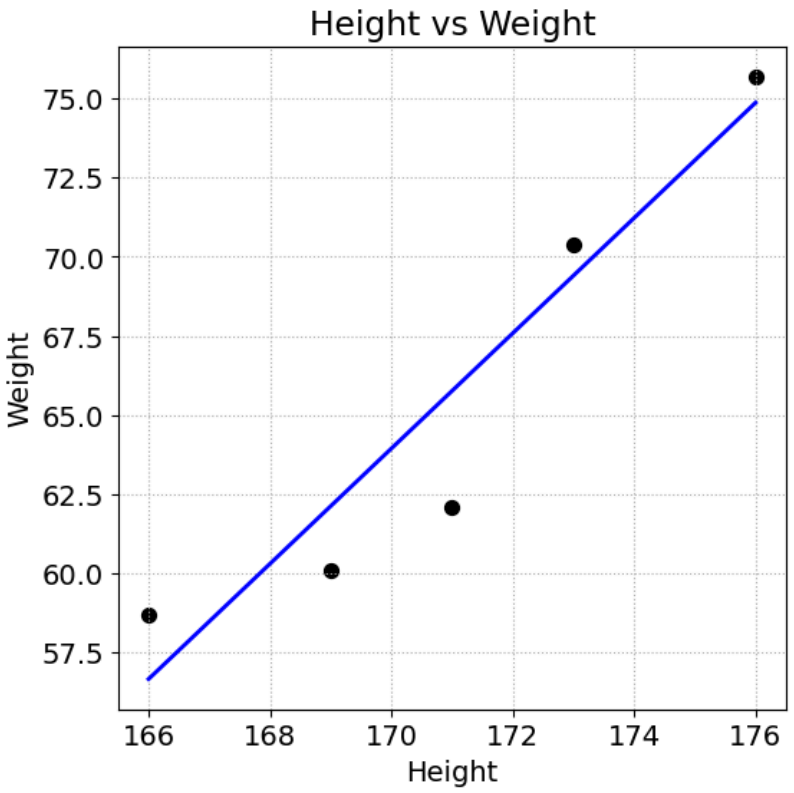
```
# 그래프 출력
#Y = W*X + B
plt.scatter(X,  Y,  c='k',  s=100)
plt.xlabel('Height')
plt.ylabel('Weight')
#plt.plot(X_range.data, Y_range.data, lw=2, c='b')
plt.plot([-5,5], [W_numpy*-5 + B_numpy, W_numpy*5 + B_numpy],c="b")
plt.title('Height vs Weight')
plt.show()
```



11. 가공 전 데이터로 회귀 직선 출력

평균값을 뺀 (X,Y)를 원래의 (x,y) 로 값을 되돌려 같은 산포도를 표시한 결과를 출력한다.

```
# y좌표와 x좌표 값 계산
# 역으로 연산 진행
x_range = X_range + x.mean()
yp_range = Y_range + y.mean()

# 그래프 출력
plt.scatter(x,  y,  c='k',  s=50)
plt.xlabel('Height')
plt.ylabel('Weight')
plt.plot(x_range, yp_range.data, lw=2, c='b')
plt.title('Height vs Weight')
plt.show()
```



<신장과 체중의 상관 직선(가공 전)>

12. 최적화 함수와 step 함수 이용

앞에서는 파라미터 W,B의 변경을 코드로 직접 수행했지만 일반적으로 ML 모델은 '최적화 함수'를 사용한다.

SGD라는 클래스의 인스턴스를 생성하고 optimizer 변수가 '최적화 함수'이다.

W,B 값을 직접 변경했던 부분을 step 함수의 호출로 최적화 함수를 이용해 파라미터의 값을 간접적으로 변경한다. 또한 경사값 초기화도 zero_grad()를 통해 구현했다.

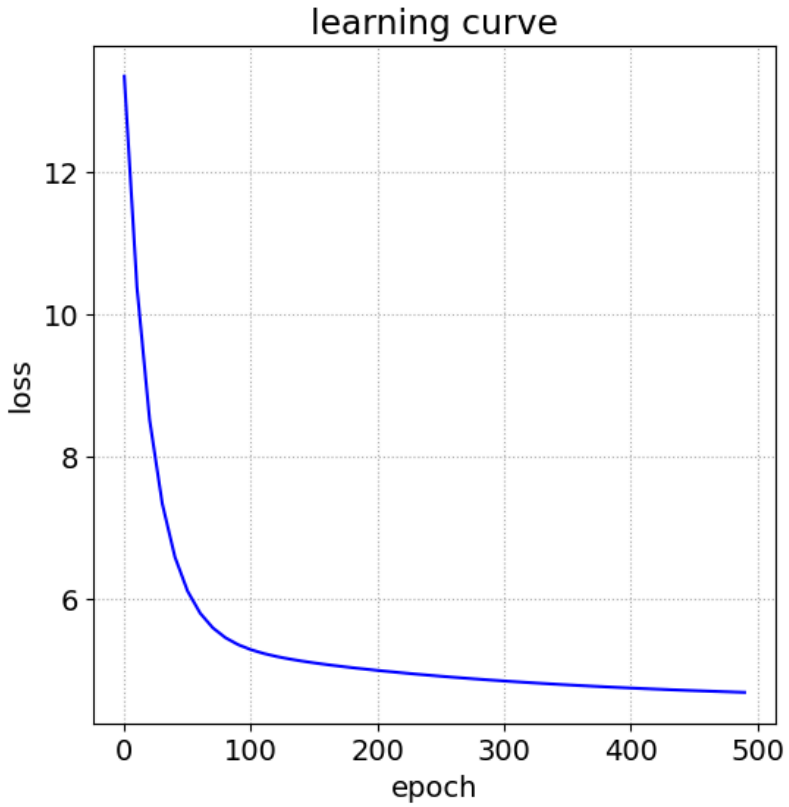
```
# 초기화

# W와 B를 변수로 사용
W = torch.tensor(1.0, requires_grad=True).float()
B = torch.tensor(1.0, requires_grad=True).float()

# 반복 횟수
num_epochs = 500
```

```
# 학습률
lr = 0.001

# optimizer 로 SGD(확률적 경사 하강법)을 사용
import torch.optim as optim
optimizer = optim.SGD([W, B], lr=lr)

# history 기록을 위한 배열 초기화
history = np.zeros((0, 2))

# 루프 처리
for epoch in range(num_epochs):

    # 예측 계산
    Yp = pred(X)

    # 손실 계산
    loss = mse(Yp, Y)

    # 경삿값 초기화
    optimizer.zero_grad()

    # 경사 계산
    loss.backward()

    # 파라미터 수정
    optimizer.step()


    # 손실 기록
    if (epoch %10 == 0):
        item = np.array([epoch, loss.item()])
        history = np.vstack((history, item))
        print(f'epoch = {epoch}  loss = {loss:.4f}')


#epoch = 0  loss = 13.3520
#epoch = 10  loss = 10.3855
```

```
#epoch = 20  loss = 8.5173
#...
#epoch = 480  loss = 4.6854
#epoch = 490  loss = 4.6796
```

```
# 최종 파라미터 값
print('W = ', W.data.numpy())
print('B = ', B.data.numpy())

# 손실 확인
print(f'초기상태 : 손실:{history[0,1]:.4f}')
print(f'최종상태 : 손실:{history[-1,1]:.4f}')

#W =  1.820683
#B =  0.3675114
#초기상태 : 손실:13.3520
#최종상태 : 손실:4.6796
```

```
# 학습 곡선 출력(손실)

plt.plot(history[:,0], history[:,1], 'b')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('learning curve')
plt.show()
```



>> step 함수의 로직

```
with torch.no_grad():
    # 파라미터 수정
    # 프레임워크를 사용하는 경우는 step 함수
    #가 이를 대신함
    W -= lr * W.grad
    B -= lr * B.grad
```

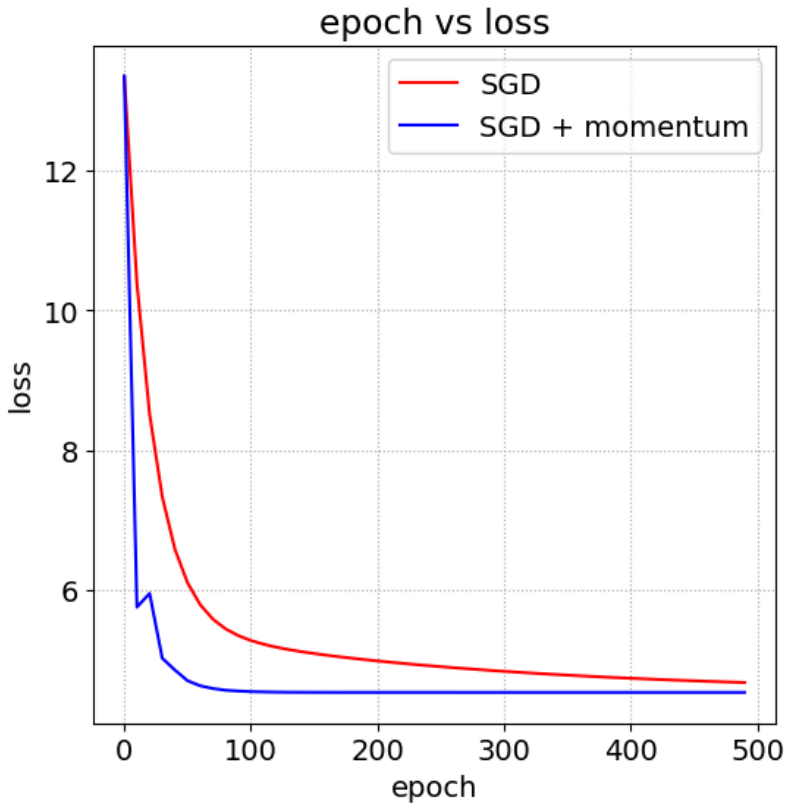
13. 최적화 함수 튜닝

momentum을 활용해서 loss 값 더욱 줄일 수 있음.

```
# 초기화

# W와 B를 변수로 사용
W = torch.tensor(1.0, requires_grad=True).float()
B = torch.tensor(1.0, requires_grad=True).float()
```

```
# 반복 횟수
num_epochs = 500

# 학습률
lr = 0.001

# optimizer로 SGD(확률적 경사 하강법)을 사용
import torch.optim as optim
optimizer = optim.SGD([W, B], lr=lr, momentum=0.9) #Optimizer

# history 기록을 위한 배열 초기화
history2 = np.zeros((0, 2))

# 루프 처리

for epoch in range(num_epochs):

    # 예측 계산
    Yp = pred(X)

    # 손실 계산
    loss = mse(Yp, Y)

    # 경사 계산
    loss.backward()

    # 파라미터 수정
    optimizer.step()

    # 경삿값 초기화
    optimizer.zero_grad()

    # 손실 기록
    if (epoch %10 == 0):
        item = np.array([epoch, loss.item()])
        history2 = np.vstack((history2, item))
        print(f'epoch = {epoch}  loss = {loss:.4f}')
```

```
# 학습 곡선(손실) 출력

plt.plot(history[:,0], history[:,1], 'r', label='SGD')
plt.plot(history2[:,0], history2[:,1], 'b', label='SGD + momentum')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.title('epoch vs loss')
plt.show()
```



튜닝한 경우 학습 속도가 빨라졌음을 확인할 수 있다.

SGD 최적화 함수 클래스에는 momentum으로 불리는 학습을 빠르게 해주는 알고리즘이 구현되어 있다.